Escola Politècnica Superior
d'Enginyeria de Vilanova i la Geltrú
UNIVERSITAT POLITÈCNICA DE CATALUNYA

PROJECTE FI DE CARRERA

TÍTOL: ESTUDI DE VIABILITAT DE LA TECNOLOGIA CUDA PER L'ACCELERACIÓ DE PROCESSOS

AUTOR: RUBÉN SÁNCHEZ CASTELLANO

TITULACIÓ: ENGINYERIA TÈCNICA EN INFORMÀTICA DE GESTIÓ

DIRECTOR: BERNARDINO CASAS FERNÁNDEZ

DEPARTAMENT: LLENGUATGES I SISTEMES INFOMÀTICS

DATA: 2011/12-Q1

TÍTOL: ESTUDI DE VIABILITAT DE LA TECNOLOGIA CUDA PER L'ACCELERACIÓ DE PROCESSOS

COGNOMS: SÁNCHEZ CASTELLANO **NOM: RUBÉN**

TITULACIÓ: ENGINYERIA TÈCNICA EN INFORMÀTICA

ESPECIALITAT: GESTIÓ **PLA: 92**

DIRECTOR: BERNARDINO CASAS FERNÁNDEZ

DEPARTAMENT: LLENGUATGES I SISTEMES INFORMÀTICS

QUALIFICACIÓ DEL PFC

TRIBUNAL

PRESIDENT **SECRETARI** **VOCAL**

DATA DE LECTURA:

Aquest Projecte té en compte aspectes mediambientals: ■ Sí □ No

PROJECTE FI DE CARRERA

RESUM (màxim 50 línies)

Aquest projecte estudia la viabilitat de l'ús de la tecnologia CUDA del fabricant NVIDIA per a accelerar l'execució de processos que tracten un gran volum de dades. Durant el desenvolupament del projecte s'analitza els cost de la modificació del codi font original per a CPU del software per adaptar-lo a la tecnologia CUDA adaptant dues aplicacions a aquesta tecnologia: la biblioteca de vídeo libvpx i el xifrador AES. A més a més, també s'analitza el temps d'execució i el consum d'energia de la versió CUDA de cada aplicació comparat amb la versió original per a CPU, i els costos del propi hardware per determinar si la tecnologia CUDA és viable econòmicament, mediambientalment i segons la dificultat de la transformació del codi font de les aplicacions.

Paraules clau (màxim 10):

CUDA	AES	Libvpx	Paral·lelització
Acceleració	Windows	GNU/Linux	Xifrat
Vídeo			

Agraïments

En primer lloc vull agrair al meu tutor la seva valentia en acceptar la direcció d'un projecte basat en una tecnologia desconeguda per ell, la seva comprensió, i tot el seu suport. Però també vull agrair a les següents persones:

Als meus pares José i Mª Teresa, pel seu suport moral durant els temps més difícils de la carrera i per tot el que he après d'ells en aquesta vida.

A Jennifer per donar-li un gir sobtat i molt positiu a la meva vida, i per tot el seu suport moral.

A tots els companys del Centre de Càlcul de l'EPSEVG, per tot el que he après d'ells i tots els moments (divertits i no tant) que hem viscut plegats.

A tots els meus amics, que m'han vist desaparèixer durant bona part de la realització d'aquest projecte.

Dedico aquest projecte als meus avis, als que hi són i als que no hi són, per tot l'afecte que he rebut d'ells durant tota la meva vida i per tot el que m'han ensenyat.

Índex

Índex de figures

Llista d'algorismes

Índex de taules

Capítol 1

Introducció

1.1 Presentació

Aquesta memòria pertany al projecte final de carrera "Estudi de viabilitat de la tecnologia CUDA per l'acceleració de processos", corresponent a la titulació d'Enginyeria Tècnica en Informàtica de Gestió. El desenvolupament d'aquest projecte explica tot el procés que s'ha seguit, des del seu plantejament fins arribar a les conclusions, passant pel desenvolupament del software i les proves sobre aquest.

1.2 Objectius

Aquest projecte té present assolir diferents objectius relacionats amb l'estudi de la viabilitat de la tecnologia CUDA per a l'acceleració de processos. Els objectius són:

- Estudiar la paral·lelització dels algorismes de la biblioteca libvpx i el xifrador AES.
- Crear aplicacions CUDA per a la biblioteca libvpx i el xifrador AES.
- Analitzar el consum d'energia i temps d'execució de les versions originals i les versions CUDA de la biblioteca libvpx i el xifrador AES.
- Estudi dels costos de hardware i del consum energètic.
- Concloure la viabilitat de la tecnologia CUDA per a l'acceleració de processos.

1.3 Motivacions

En el moment de decidir la temàtica del projecte vaig tenir en compte diferents idees. Sempre he sigut una persona interessada pel funcionament de les màquines, és per això que el meu interès dintre de la programació es decanta més pel desenvolupament de software per a altres dispositius, no només en el desenvolupament de software per a ordinador. Un exemple és un treball dirigit que vaig realitzat que implementa el protocol de transferència trivial de fitxers (*TFTP*) en la plataforma de la consola portàtil Nintendo DS.

També participo en projectes de computació distribuïda a través de la plataforma BOINC, és en aquesta plataforma on vaig veure per primer cop una aplicació de la tecnologia CUDA. Un dels projectes que usen la plataforma BOINC és SETI@Home[17] i el seu equip de desenvolupadors van fer publica una versió CUDA del seu programa que reduïa el temps de càlcul de les tasques fins a una tercera part usant una GPU NVIDIA 8800GTS (GPU que tenia abans

de la usada per a aquest projecte). Va ser en aquest moment on va néixer el meu interès per aquesta tecnologia.

Prèviament a la decisió d'escollir la tecnologia CUDA com a base del projecte, vaig tenir en compte una altra alternativa. En els últims mesos, l'interès pel món dels telèfons intel·ligents ha crescut en gran mesura. Com a aficionat a la tecnologia també he estat atret a aquest món. La assistència al Mobile World Congress 2011 i la meva participació a l'organització de l'edició 2011 del Genmob: The Smartphone Day, han acabat assentant l'interès pel desenvolupament d'aplicacions per a dispositius mòbils. Més concretament, el meu interès dintre aquest món es centra a la plataforma Android.

Però, encara que ja he desenvolupat alguna aplicació per a Android abans de prendre la decisió de desenvolupar aquest projecte entorn a la plataforma CUDA, el motiu que em va decantar per escollir aquesta segona temàtica va ser, precisament, la gran popularitat d'Android envers la poca popularitat que té la tecnologia CUDA. Aquest últim motiu, juntament amb la meva afició cap al desenvolupament de software per a dispositius, és el que em va motivar el desenvolupament d'aquest projecte.

1.4 Estructura de la memòria

Com ja s'ha explicat anteriorment, aquesta memòria analitza tot el procés seguit per assolir els objectius esmenats a l'apartat anterior segons l'estructura que es redacta a continuació:

Capítol 2 Conté la descripció detallada del projecte, on s'expliquen la problemàtica que s'intenta resoldre, com es planteja la seva resolució i amb quins mitjans es farà.

Capítol 3 Planteja la planificació inicial del projecte. En aquest capítol s'exposa en quines tasques es divideix el treball d'aquest projecte i quina és la duració estimada de cada tasca, així com les dates d'inici i fi de cadascuna.

Capítol 4 Aquest capítol conté l'avaluació tecnològica del projecte, quines tecnologies s'han plantejat, les seves característiques, quines s'han escollit i quines no, i perquè de la seva selecció o rebuig.

Capítol 5 Conté una explicació del funcionament del llenguatge CUDA C/C++ exemplificat.

Capítol 6 S'explica el funcionament dels algorismes de la biblioteca libvpx i el xifrador AES per tal de raonar quina és l'estratègia a seguir per a la seva acceleració. En aquest apartat, també s'explica quines han estat les modificacions fetes al codi dels algorismes per fer-los compatibles amb la tecnologia CUDA.

Capítol 7 Una vegada explicat els algorismes i el desenvolupament de les versions CUDA de tots dos algorismes, es presenten les proves que s'han fet d'aquest software per tal de comparar-los amb les seves versions originals. Tanmateix, s'estudia si les versions CUDA del software presenten avantatges sobre les versions originals o, en cas contrari, si no val la pena apostar per a aquesta tecnologia.

Capítol 8 S'exposa la planificació final i el cost del projecte.

Capítol 9 Es plantegen les conclusions del projecte i també s'explica el treball futur que es pot fer sobre aquesta tecnologia i sobre el software desenvolupat.

Capítol 2

Descripció del projecte

2.1 El problema

Des de fa uns anys fins ara, els suports per a l'emmagatzematge de les dades han evolucionat molt. Avui en dia, és normal trobar discos durs, memòries flash o suports òptics que fàcilment suporten capacitats superiors al gigabyte. Això és degut a l'augment de la mida dels continguts multimèdia, que en els darrers anys han vist millorada la seva qualitat d'una manera molt notable a la indústria de l'entreteniment, com per exemple les pel·lícules o videojocs en alta definició.

Aquesta evolució envers la qualitat dels continguts multimèdia té com a efecte col·lateral un augment en la mida dels arxius que els contenen. En els suports digitals existents, una millor qualitat d'imatge vol dir tenir molta més informació del contingut, és a dir, una quantitat més gran de dades. El problema es presenta quan volem processar aquests continguts en alta definició.

Si bé els continguts multimèdia amb menys qualitat que l'alta definició es poden processar amb un cost raonable (en temps i espai) gràcies a l'evolució del hardware de les CPU, no passa el mateix amb el contingut en alta definició pel problema abans esmenat. Encara que s'han desenvolupat solucions per reduir el cost espacial d'aquests continguts no s'ha fet el mateix amb el cost temporal que suposa tractar-los per reduir el primer dels costos. Es a dir, tractar un volum de dades com els dels actuals continguts en alta definició té un cost molt més elevat, temporalment parlant, que els continguts més antics i que no tenen tanta qualitat d'imatge.

Encara que la potència de càlcul de les CPU ha millorat molt i a més disposen de diversos nuclis, tractar aquestes dades segueix essent una feina costosa temporalment. Llavors, existeix alguna manera de reduir aquest cost temporal de tractar els continguts multimèdia amb el hardware existent en un ordinador domèstic?

2.2 Una possible solució: la GPU

Un dels components dels ordinadors domèstics de sobretaula és la targeta gràfica, el dispositiu encarregat de presentar una interfície visual a l'usuari final. Aquest component té la responsabilitat de dibuixar tota la informació que li arriba al monitor, per poder fer això conté un processador i una memòria RAM encarregats de fer tota aquesta feina. Fins fa ven poc, el processador d'una targeta gràfica era de propòsit específic, només podia dibuixar, però al llarg del últims anys la necessitat de mostrar millors gràfics ha forçat a crear targetes gràfiques molt més potents i amb més memòria. Ha arribat un punt en el que aquests components són molt més potents que la CPU.

Arribats a aquesta situació, els principals fabricants d'aquests components van començar a adaptar les targetes gràfiques per a que poguessin fer molt més que dibuixar al monitor i

així aprofitar la seva potència per a realitzar altres feines. És llavors quan es comença a parlar de les GPGPU . Aquestes unitats de processament hereten de les GPU la característica de poder executar moltes instruccions al mateix temps. Així que, amb les GPGPU es disposa d'un dispositiu que pot executar un número de tasques a la vegada (i gairebé qualsevol tipus) amb una gran memòria RAM.

2.2.1 Plataformes hardware disponibles

Actualment existeixen dues empreses que són els principals desenvolupadors de GPUs, són la nord-americana NVIDIA i la canadenca ATI.

2.2.1.1 NVIDIA

NVIDIA és una de les grans empreses dintre del món de la informàtica, especialitzada en la fabricació de targetes gràfiques, controladors de plaques base, i durant els últims anys també de CPU per a dispositius mòbils. L'any 2006, va presentar una nova tecnologia incorporada a les seves GPUs que permetia usar-les com a processadors de propòsit general anomenada NVIDIA CUDA.

Com a casos d'èxit de CUDA podem trobar que s'usa en centres de supercomputació, com en el Barcelona Supercomputing Center [2], i en projectes de recerca científica com el projecte SETI@Home. A més, ja existeixen aplicacions de software propietari que usen aquest potencial per a la compressió de vídeo, com el programa Badaboom [8]. A la figura 2.1 es pot veure l'estructura d'aquesta plataforma, els diferents llenguatges de programació suportats, les biblioteques desenvolupades per a aquesta plataforma i les dues grans arquitectures actuals dels dispositius que implementen CUDA.

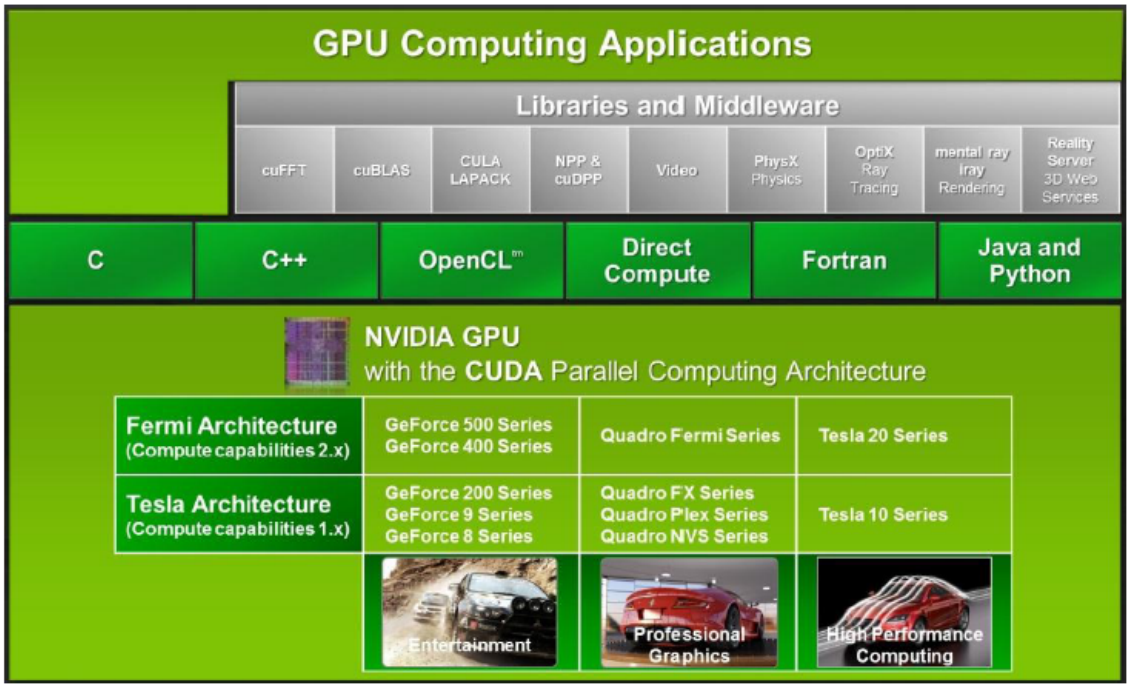

Figura 2.1: Estructura de l'entorn CUDA[16]

2.2.1.2 ATI

ATI és l'altre gran fabricant de GPUs avui en dia, competidor directe de NVIDIA en hardware d'acceleració gràfica. En el seu cas van trigar una mica més en presentar una tecnologia que

permetés usar les GPU d'aquesta casa per a una computació de propòsit general. No va ser fins l'any 2008 que ATI va presentar aquesta tecnologia, anomenada ATI Stream. L'Agost de 2011 es va canviar el nom d'aquesta tecnologia per passar a dir-se ATI APP.

A l'igual que passa amb NVIDIA CUDA, ATI APP s'usa en supercomputadors com la TianHe-1[5] i en projectes de recerca científica com Folding@Home[20].

2.2.1.3 El hardware escollit

Tot i que es poden usar qualsevol de les dues plataformes hardware per complir l'objectiu d'aquest projecte, aquests dispositius tenen un cost econòmic notable. Tenint en compte això i que es disposa d'un hardware amb la tecnologia NVIDIA CUDA, s'opta per aquesta plataforma.

2.3 La biblioteca libvpx

Per tal de poder veure un exemple de l'aplicació de la tecnologia CUDA per a l'acceleració de processos a la problemàtica del contingut multimèdia, farem servir una biblioteca per a descodificar els continguts multimèdia.

De biblioteques d'aquest tipus hi ha un gran nombre, però per facilitar el desenvolupament de l'aplicació s'ha optat per una que sigui software lliure. Això implica, a grans trets, que aquesta biblioteca es pot usar lliurement, modificar, redistribuir, i redistribuir les modificacions que s'hagin fet d'aquesta. Una de les biblioteques que compleixen aquests criteris és la biblioteca libvpx, encara que existeixen d'altres biblioteques de vídeo com la biblioteca x268[19]. Aquesta biblioteca és l'alternativa de software lliure a la biblioteca propietària H.264, una biblioteca per a tractar vídeo en alta definició al igual que libvpx. També es van tenir en compte d'altres codificadors de vídeo com: MPEG-2[12], o Xvid[30], no són codificadors per a vídeo en alta definició però també podrien ser usat per a aquest projecte. Però totes aquestes opcions no s'han descartat pel gran suport documental i de la comunitat que té libvpx.

La biblioteca libvpx era propietat de l'empresa On2, però l'Agost de l'any 2009 Google Inc. la va comprar. Tot seguit va alliberar el codi del còdec VP8 que On2 estava desenvolupant sota una llicència de software lliure, la llicència BSD. A data del començament d'aquest projecte (1 de Setembre de 2011), l'última versió disponible d'aquesta biblioteca és la 0.9.7-p1, i és per això que és l'escollida per a aquest projecte.

2.4 El xifrador AES

A banda de la biblioteca libvpx, per tal de veure un altre exemple de l'aplicació de la tecnologia CUDA, s'ha decidit desenvolupar una versió CUDA del xifrador AES.

El xifrador AES (*Advanced Encryption Standard*), que deriva del xifrador Rijndael (pels seus autors: Joan Daemen i Vincent Rijmen), és un xifrador per blocs que es va convertir l'any 2002 en un estàndard. Des de l'any 2006, AES és el xifrador per bloc simètric més popular arreu del món.

Aquest estàndard va sorgir d'un concurs fet per l'Institut Nacional de Normes i Tecnologia (NIST) l'any 1997. L'objectiu d'aquest concurs era trobar un algorisme de xifrat suficientment segur per a protegir les dades que fos resistent a atacs per força bruta i per criptoanàlisi, a més a més substituiria als algorismes anteriors com DES que ja eren dèbils envers atacs amb força bruta. Els objectius d'aquell concurs els compleix gràcies a la longitud de les claus que pot usar (128, 192 i 256 bits) i per la pròpia estructura del algorisme.

Degut a la simplicitat de l'estructura d'aquest algorisme és pot implementar tant en hardware com en software sense utilitzar una gran quantitat de recursos en la seva execució. És per

aquest motiu que avui en dia podem trobar implementacions d'AES en llocs tan quotidians com per exemple:

- Xip de les targes de crèdit.
- Accés segur a pàgines web i comunicacions per Internet mitjançant SSL.
- Encriptació de comunicacions inalàmbriques WIFI.
- Xifrat de dades amb diferents compressors d'arxius.

A més a més, aquest algorisme de xifrat ha estat estudiat a l'assignatura optativa de Criptografia. Per tant, donat que es tenen coneixements sobre el disseny d'aquest algorisme el fan un candidat perfecte per a fer-ne una versió CUDA. També cal dir, que el codi font de la biblioteca libvpx és un codi desconegut per a l'autor, i això no assegura, en el moment de decidir quin software s'implementaria per a exemplificar l'ús de CUDA, que la versió CUDA de la biblioteca arribi a ser completament funcional. Així, amb el xifrador AES es té molta més seguretat de desenvolupar un software CUDA completament funcional.

2.4.1 Entorn hardware

A la següent llista es detallen tots els components de l'entorn hardware de que es disposa i que s'usarà en el desenvolupament i testeig de l'aplicació:

- Placa base : Asus P5WDH Deluxe
- CPU Intel Core2Duo. Les dades d'aquest model són les que es mostren amb el software CPU-Z a la figura .

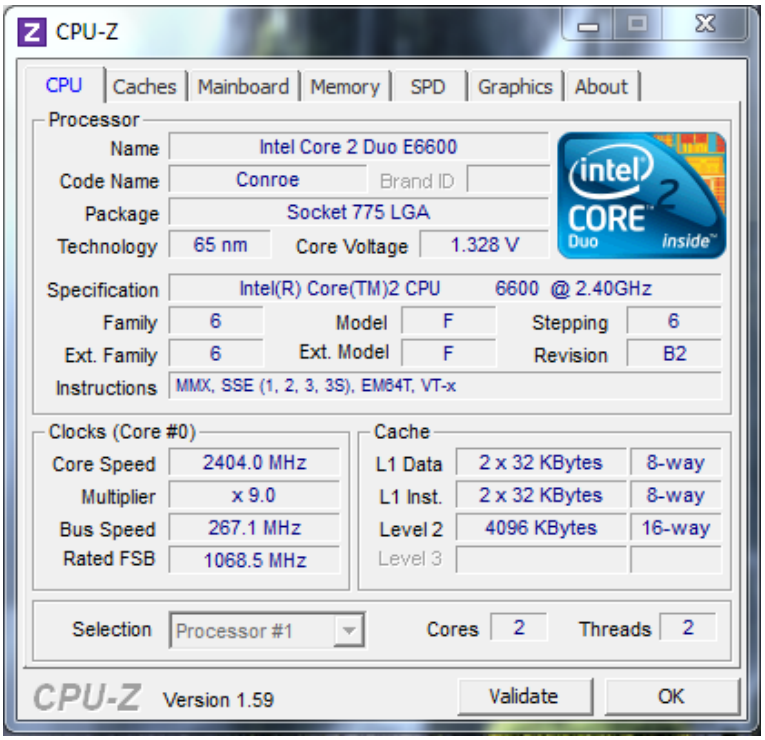

Figura 2.2: Dades del model Core 2 Duo E6600 d'Intel

- 2GB Memòria RAM DDR2-800 DualChannel

- Disc dur: Seagate Barracuda SATA2 de 7200 rpm.
- Targeta gràfica: Asus GTX550Ti
  - Memòria total disponible: 985 MB
  - Rellotge de la GPU: 1.82GHz
  - Número màxim de fils d'execució per bloc: 1024
  - Mida màxima de cada dimensió del bloc: 1024 x 1024 x 64
  - Mida màxima de cada dimensió de la graella: 65535 x 65535 x 65535
  - Velocitat de transferència Hoste -> Dispositiu: 1238.9 MB/s
  - Velocitat de transferència Dispositiu -> Hoste: 1092.1 MB/s
  - Velocitat de transferència Dispositiu -> Dispositiu: 44823.7 MB/s

  Moltes d'aquestes característiques s'expliquen al capítol dedicat a la descripció en detall de la tecnologia CUDA.

Per altra banda cal dir que la CPU i GPU emprades en aquest projecte implementen tecnologies que tenen uns cinc anys de diferència. Per tal de fer una comparació el més justa possible, també es provarà la versió CPU de les aplicacions en un Intel Core i7 920 que, aproximadament va sortir al mercat al mateix any que la GPU usada. A la figura 2.2es poden veure les dades sobre aquesta CPU. Cal dir que aquesta CPU té característiques més avançades com multi-threading o l'opció de deshabilitar diferents nuclis, però en les proves fetes s'han deixat els paràmetres de fàbrica ja que l'objectiu és comparar les dues CPU tal i com es compren a la botiga.

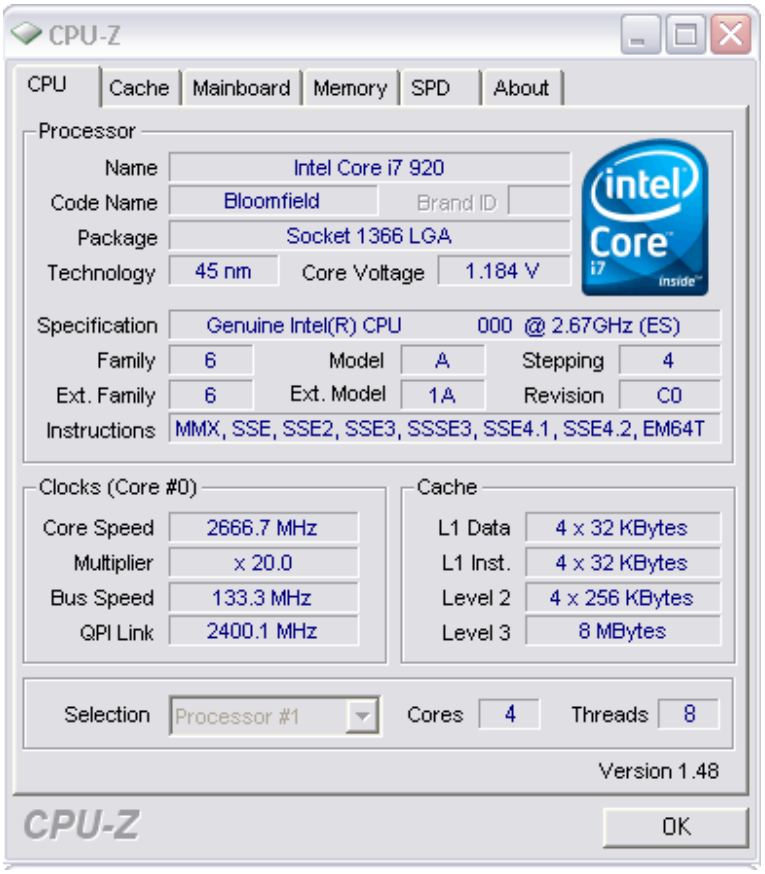

Figura 2.3: Dades del model Core i7 920 d'Intel[22]

Capítol 3

Avaluació tecnològica

3.1 El llenguatge de programació

Dintre de CUDA existeixen compiladors per a diferents llenguatges de programació, així que a priori és indiferent quin utilitzar. A continuació es llisten els llenguatges més coneguts.

3.1.1 CUDA C/C++

El llenguatge de programació CUDA C/C++ és una extensió del llenguatge C/C++ amb unes certes limitacions. En aquest llenguatge es disposen de totes les funcionalitats de C/C++ a les que s'afegeixen les instruccions necessàries pel desenvolupament d'aplicacions CUDA i les extensions dels arxius que contenen el codi font són *.cu* per a la implementació i *.cuh* per a les capçaleres.

Avantatges:

- Basat en C/C++, molt estès i conegut.
- Coneixements previs de C/C++, adaptació més ràpida al llenguatge.
- La biblioteca libvpx està implementada en aquest mateix llenguatge.
- El llenguatge amb millor suport per part de NVIDIA.
- Llenguatge multi-plataforma.

Inconvenients:

- Baix nivell d'abstracció.
- Restriccions al llenguatge C/C++.

3.1.2 OpenCL

OpenCL és un framework per a la computació en paral·lel. Aquest framework ha estat creat per a poder crear aplicacions que puguin ser executades en qualsevol tipus d'unitat de procés. És a dir, una aplicació desenvolupada en OpenCL es pot fer servir en CPUs, GPUs entre d'altres tipus de processadors sempre i quan aquests processadors siguin compatibles amb aquest llenguatge. Actualment, Khronos Group és la entitat encarregada de la gestió del estàndard OpenCL, encara que inicialment va ser creat per Apple.

Avantatges:

- Basat en C99 , dialecte de C.
- Gran compatibilitat amb hardware diferent.
- Coneixements previs de C/C++, adaptació més ràpida al llenguatge.

Inconvenients:

- Baix nivell d'abstracció, encara més que CUDA C/C++.
- Existeix menys documentació de NVIDIA sobre OpenCL.

3.1.3 C# i Microsoft Direct Compute

La solució de la nord-americana Microsoft per a desenvolupar aplicacions per a executar sobre GPGPUs passa pel llenguatge C# i la API Microsoft Direct Compute.

Avantatges:

- Basat en C#.
- Alt nivell d'abstracció.
- Bon suport per part de Microsoft.

Inconvenients:

- Manca de coneixements sobre C#.
- Manca de documentació per part de NVIDIA.
- Nul·la portabilitat del codi a altres plataformes. Només és compatible amb Windows Vista i Windows 7.

3.1.4 Java, Python i Fortran

Des de l'aparició de CUDA s'han desenvolupat diferents wrappers desenvolupats per terceres parts que fan possible utilitzar les característiques de CUDA amb altres llenguatges de programació. Ja que aquests llenguatges no són suportats pel fabricant del dispositiu CUDA, no es tenen compte en la decisió final del llenguatge.

3.1.5 El llenguatge escollit

Vistes totes les opcions possibles dintre dels llenguatges de programació compatibles amb CUDA s'ha escollit C/C++ pels següents motius:

- Coneixements previs del llenguatge.
- Compatibilitat directa amb la biblioteca 'libvpx', que està escrita en el mateix llenguatge.
- Portabilitat del codi a altres sistemes operatius.
- Pel suport que ofereix NVIDIA cap a aquest llenguatge en concret (eines i documentació).

No s'ha escollit OpenCL per la poca documentació existent. Tampoc s'ha escollit cap llenguatge de la resta per que són diferents al llenguatge amb el que s'implementa la biblioteca 'libvpx' o bé perquè el codi resultant seria poc portable cap a altres plataformes software (en el cas de C# i Microsoft Direct Compute).

3.2 Desenvolupament i proves

3.2.1 Entorn de desenvolupament

Aquest projecte planteja una solució multi-plataforma per a sistemes operatius Microsoft Windows i GNU/Linux per a analitzar el grau d'adaptació del codi CUDA a diferents sistemes operatius. No es desenvolupa una solució per a més plataformes com podria ser Mac OS X, per falta de les llicències d'aquests sistemes operatius, i del hardware necessari. Primer analitzarem les possibilitats envers les versions d'aquests dos sistemes operatius.

La plataforma Windows gaudeix d'una molt bona posició al mercat, per això i pel suport de NVIDIA s'ha escollit aquesta com a una de les plataformes de desenvolupament. Actualment hi predominen tres versions de Windows al mercat: Microsoft Windows XP, Microsoft Windows Vista i Microsoft Windows 7. Ja que Microsoft i NVIDIA donen suport preferent a Microsoft Windows 7, i es disposa de la llicència d'aquest, s'ha escollit com a sistema operatiu per al qual desenvolupar aquest projecte.

L'altra gran sistema operatiu present avui en dia és GNU/Linux. D'aquests sistema operatiu existeixen moltes distribucions diferents, cadascuna implementa el sistema operatiu d'una forma diferent, però el nucli (kernel) Linux es manté a totes. Una de les distribucions GNU/Linux més famoses ara per ara és Ubuntu Linux. A més a més de la seva popularitat, s'ha escollit aquesta distribució pel suport de la comunitat, i el suport de NVIDIA per aportar una versió del kit de desenvolupament per a aquesta distribució. Concretament, NVIDIA distribueix un paquet per la versió 10.10 d'aquesta distribució, així que Ubuntu 10.10, nom en clau Maverick Meerkat és el segon sistema operatiu objectiu de l'aplicació d'aquest projecte.

NVIDIA posa a disposició de tothom a la seva web de les eines necessàries per a crear aplicacions CUDA. Aquestes eines inclouen el compilador, documentació i aplicacions d'exemple. Aquestes eines es distribueixen en el que s'anomena kit de desenvolupament d'aplicacions, i es poden descarregar versions d'aquests kits per a Windows, distribucions GNU/Linux entre d'altres sistemes operatius. D'aquesta manera, el desenvolupador té més llibertat per poder escollir sobre quin sistema operatiu desenvolupar la seva aplicació CUDA. Així que l'únic que a de fer el programador és trobar una eina d'edició de codi, i juntament amb el compilador del kit de desenvolupament ja pot crear les seves aplicacions CUDA. Del kit de desenvolupament s'usarà la versió 4.0 per ser l'última presentada per NVIDIA a dates de la realització d'aquest projecte tant a Microsoft Windows 7 com a Ubuntu 10.10.

3.2.1.1 Windows 7

Per al sistema operatiu de Microsoft hi ha una infinitat d'editors de codi, però el més conegut és el propi: Microsoft Visual Studio. Es podria usar un altre editor qualsevol, però el motiu clau que motiva l'elecció d'aquest en concret és el fet que el compilador que NVIDIA distribueix dins el kit de desenvolupament per a CUDA necessita del compilador de Microsoft Visual Studio 2008. Es podria escollir qualsevol altre editor de codi per desenvolupar l'aplicació, però caldria instal·lar el compilador de Visual Studio igualment. Per aquest motiu, i per la quantitat d'informació que existeix sobre aquest IDE s'ha optat per escollir una versió específica de Visual Studio per a C/C++, concretament: Microsoft Visual C++ 2008 Express[14]. No s'usa la versió de 2010 ja que la seva integració amb el kit de desenvolupament de CUDA dóna problemes. Per altra banda, s'ha intentat usar Netbeans a Windows per així usar el mateix entorn que a Ubuntu, però per la dificultat de integrar-lo amb CUDA a Windows i l'existència de Visual Studio per a aquesta plataforma es va descartar.

3.2.1.2 Ubuntu 10.10 Maverick Meerkat

A la distribució de GNU/Linux Ubuntu[13] també existeixen un munt d'eines de desenvolupament, de fet el kit de desenvolupament de CUDA per a aquest sistema operatiu usa el compilador GCC present al sistema de forma predeterminada, així que en principi es pot usar qualsevol editor de codi per desenvolupar una aplicació CUDA. Però per estalviar feina en el moment de desenvolupar l'aplicació, s'usarà un IDE, i un dels més famosos és Netbeans[1].

Per a aquest editor existeix un plugin que configura l'editor per a que CUDA s'integri de forma automàtica. Aquest plugin, juntament amb l'experiència prèvia de l'autor amb aquesta eina i el suport actualment existent de la comunitat el fan la millor opció. Concretament, s'ha escollit la versió 6.9 de Netbeans ja que és la suportada oficialment a la versió usada d'Ubuntu.

3.2.2 Entorn de proves i testeig de les aplicacions

Les dues aplicacions d'aquest projecte es desenvoluparan inicialment en Ubuntu, així com les proves i la depuració del software. Una vegada el software estigui depurat i funcioni correctament, es compilarà a Windows 7 per a veure així la compatibilitat del codi. Per una altra banda, la diferència entre les dues aplicacions que es desenvoluparan obliguen a que les proves siguin totalment diferents.

En el cas de la biblioteca libvpx, s'utilitzarà l'aplicació *simple_ decoder*, que és una de les aplicacions que acompanyen al SDK de la biblioteca. S'ha decidit que atés que no s'ha treballat anteriorment amb el codi font d'aquesta biblioteca, el primer desenvolupament ha de ser el més simple possible per a facilitar la depuració del codi. Cal tenir en compte que parlem de la biblioteca libvpx, però en el desenvolupament de l'aplicació no es generarà cap biblioteca, es compilarà el codi font d'aquesta juntament amb el codi font de *simple_ decoder*.

L'aplicació *simple_ decoder* és un descodificador simple que espera un arxiu en format IVF com a entrada, i com a sortida genera un arxiu de vídeo en format YUV descomprimit. Però, per a generar aquest vídeo en format IVF s'utilitzarà una altra aplicació que acompanya també al SDK de la biblioteca, l'aplicació *vpx_ enc*. Aquesta aplicació és un convertidor de vídeo amb més funcionalitats, pot comprimir i descomprimir arxius de vídeo en diferents formats, IVF entre ells.

Finalment, per veure si el vídeo generat per *simple_ decoder* és correcte, s'utilitzarà el reproductor de medis multimèdia MPlayer a través de comandes per consola. El format de vídeo YUV és un format de vídeo pla, és a dir, no conté cap mena d'informació sobre el vídeo que conté, llavors si s'intenta reproduir un vídeo en aquest format mitjançant una aplicació amb entorn gràfic, no mostrarà correctament el vídeo perquè no sabrà les mides dels fotogrames (una de les metadades que el format YUV no conté).

Respecte al segon software que es desenvoluparà en aquest projecte, el xifrador AES, les proves seran molt més simples. El desenvolupament del software també s'iniciarà a Ubuntu i una vegada depurat es traslladarà a Windows 7. En quant a les dades que s'usaran en aquest aplicatiu, seran les incloses a l'apèndix C de l'especificació de l'estàndard AES[18].

Una vegada verificat que el software funciona correctament per a una petita quantitat de dades, llavors s'executarà amb una quantitat variable de dades variable per a veure el rendiment d'ambdues versions, la versió CPU i la versió CUDA. Les dades amb les quals s'analitzarà el rendiment de l'aplicació seran de 128, 256, 384 i 512MB.

En el capítol corresponent al desenvolupament d'aquestes aplicacions s'explica detalladament tot el procés seguit en ambdues aplicacions.

Capítol 4

Planificació inicial

4.1 Tasques del projecte

Per tal d'organitzar tota la feina que comporta aquest projecte, s'ha creat una planificació del mateix dividint-lo en subtasques. Aquestes subtasques són dependents les unes de les altres, és a dir, l'inici d'una tasca requereix de la finalització de la tasca que la precedeix.

	Modo de tarea	Nombre de tarea	Duración	Comienzo	Fin
1		**PFC**	**102,75 días**	**jue 01/09/11**	**lun 09/01/12**
2		**Recopilació de tecnologies**	**22 días**	**jue 01/09/11**	**mié 28/09/11**
3		Análisi de WebM (libvpx)	10 días	jue 01/09/11	mar 13/09/11
4		Análisi de CUDA	10 días	mar 13/09/11	lun 26/09/11
5		Análisi de la GUI	2 días	lun 26/09/11	mié 28/09/11
6		Documentació	4 días	jue 01/09/11	lun 05/09/11
7		**Implementació**	**84,75 días**	**jue 01/09/11**	**sáb 17/12/11**
8		Implementació de l'aplicacions per consola	50 días	mié 05/10/11	mié 07/12/11
9		Testeig i correcció de les aplicacions per consola	8 días	mié 07/12/11	sáb 17/12/11
10		Documentació	5 días	jue 01/09/11	mié 07/09/11
11		**Redacció de la memòria**	**18 días**	**sáb 17/12/11**	**lun 09/01/12**
12		Analisi del rendiment de les versions CUDA	8 días	sáb 17/12/11	mar 27/12/11
13		Compilació de tota la documentació	10 días	mié 28/12/11	lun 09/01/12

Figura 4.1: Llistat de tasques del projecte

Com es pot veure a la figura 4.1, s'ha dividit el projecte en tres tasques principals, cal tenir en compte que a totes les subtasques es té en compte la redacció de la documentació pertinent a aquesta en la memòria del projecte encara que com és una feina que es fa mentre es desenvolupa la tasca no es té en compte a l'hora de calcular el nombre total de dies que es trigarà a fer el projecte.

4.2 Descripció de les tasques

Tot seguit, es descriuen cadascuna de les parts enumerades a la figura 4.1 en la secció anterior.

Recopilació de tecnologies Aquesta tasca es centra en preparar i conèixer totes les eines necessàries per a poder dur a terme aquest projecte. Això inclou l'estudi de la tecnologia CUDA, de la biblioteca libvpx, tenint en compte que tampoc cal invertir massa temps ja que les aplicacions no requereixen profunditzar en l'estudi d'aquestes tecnologies, només comprovar la seva viabilitat.

Implementació Inicialment, es crearan les aplicacions per a consola per a la biblioteca libvpx i pel xifrador AES. S'ha decidit no fer cap tipus de , ja que no té cap tipus de valor afegit i no ajuda a assolir l'objectiu d'aquest projecte, amb una aplicació per a consola és suficient per analitzar la viabilitat de la tecnologia CUDA. Cal dir també, que aquest projecte no té pas una etapa d'especificació o disseny del software ja que aquest projecte té com a objectiu adaptar codi ja escrit a la tecnologia CUDA.

Redacció de la memòria Una vegada les aplicacions estiguis desenvolupades i comprovades, caldrà fer els anàlisis de rendiment, on es mesurarà la diferència en els temps d'execució de les dues versions de les aplicacions i quanta energia consumeixen. Finalment, la darrera tasca que resta després de l'anterior, és compilar tota la documentació i redactar la memòria final del projecte.

4.3 Resum

La planificació inicial d'aquest projecte té en compte les tasques de la figura 4.1 i també les seves dates. Així doncs, la data d'inici del projecte és el dia 1 de Setembre de l'any 2011 i s'estima finalitzar-ho el dia 9 de Gener de 2012. Cal tenir en compte que per a realitzar aquesta planificació, s'ha establert el següent horari per a dedicar a treballar el projecte:

- De dilluns a divendres de 16.00 hores fins a les 22.00 hores.
- Caps de setmana de 10.00 a 13.00 i de 15.00 a 19.00 hores.

Capítol 5

CUDA

5.1 Introducció a CUDA

Degut a la demanda del mercat de consum del processament de continguts 3D i d'alta definició, com ja s'ha explicat al capítol 2, s'han transformat els processadors gràfics en processadors multi-nucli amb una gran capacitat de paral·lelització amb una enorme capacitat de càlcul i amb un accés a memòria molt ràpid. Com es pot veure a la figura 5.1 la capacitat de càlcul d'aquest dispositius comparada amb la de les CPUs és molt superior, això també es pot veure en l'ample de banda a l'hora de processar les dades, com es veu a la figura 5.2.

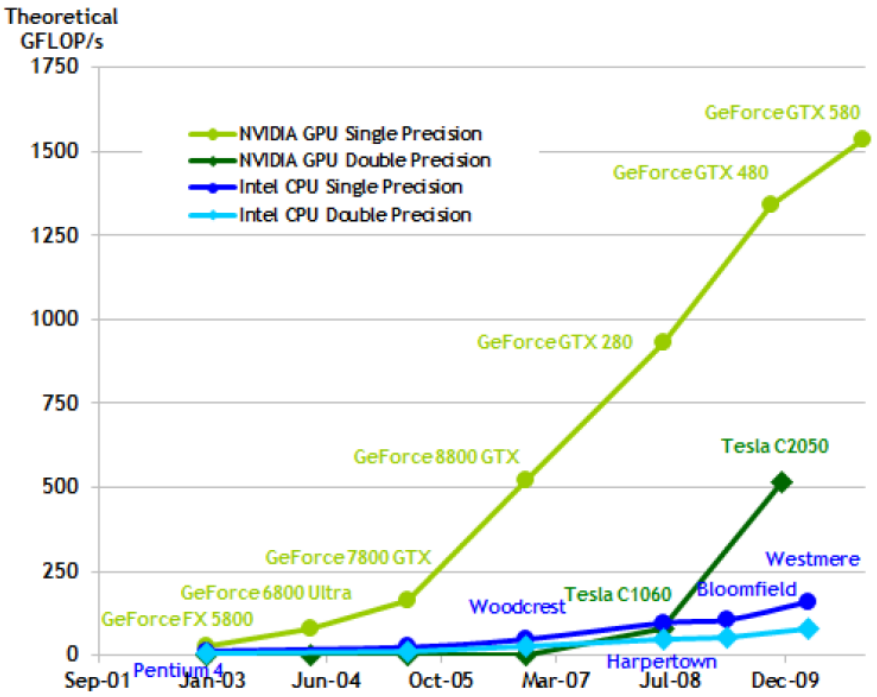

Figura 5.1: Comparació de la potència de càlcul entre diferents models de CPU i GPU[16]

5.1.1 Arquitectura

Per entendre una mica millor com funcionen les GPGPUs de NVIDIA, a continuació es fa una petita introducció a la seva arquitectura hardware.

La diferència de rendiment entre les GPGPUs i les CPUs és degut a que les primeres estan preparades per a un tractament intensiu de les dades i amb una alta paral·lelització (això és el

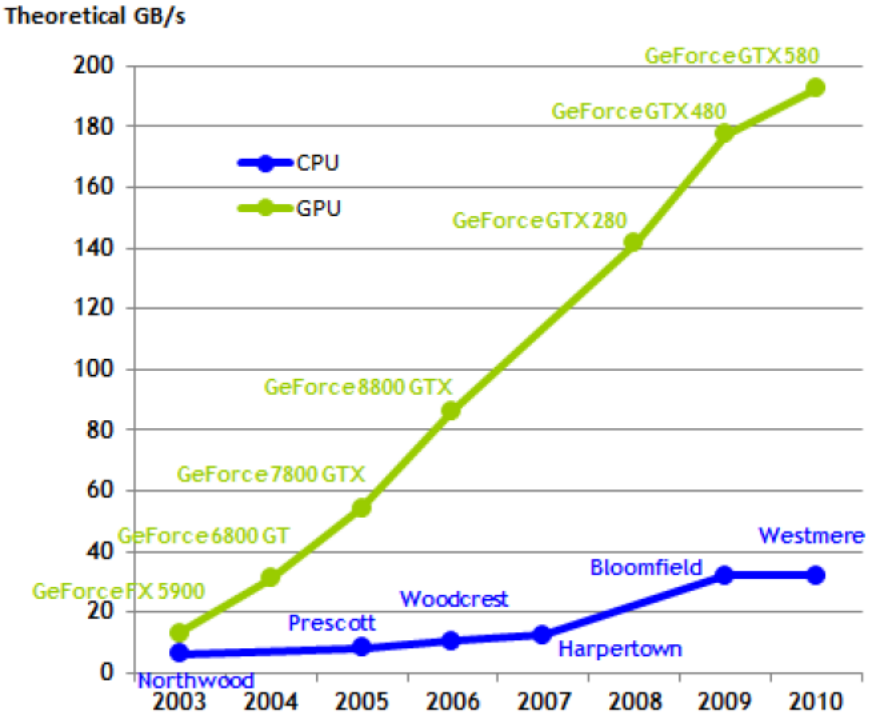

Figura 5.2: Comparació de l'ample de banda d'accés a memòria entre diferents models de CPU i GPU[16]

que requereix, precisament, el dibuixat del gràfics que fan aquest tipus de dispositius), per això el hardware conté moltes més unitats de processament de dades que no pas per al control de flux o l'emmagatzematge de les dades a la caché del dispositiu. Aquesta idea es pot veure fàcilment representada a la figura 5.3, on es pot veure una CPU amb quatre nuclis de processament controlats per una única unitat de control i una unitat de caché. En canvi a la dreta es pot veure una GPU amb moltes més unitats de processament per a cada parell d'unitat de control i de caché. A més a més, en aquesta imatge la DRAM representa tota la memòria tant de la CPU com de la GPU, més endavant s'entra més en detall en l'estructura de la memòria que té un dispositiu CUDA.

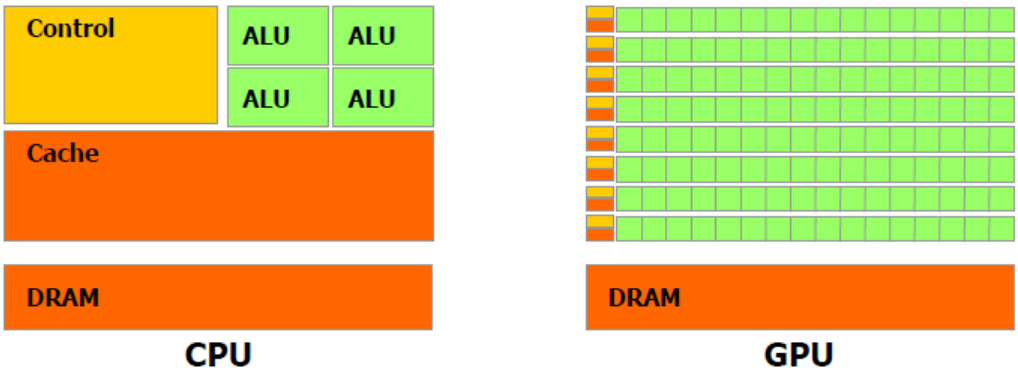

Figura 5.3: Comparació d'arquitectures CPU i GPU[16]

Les GPU són dispositius pensats per a resoldre problemes que es poden expressar com a càlculs en paral·lel, és a dir, on un programa executa el mateix algorisme en diferents blocs de dades independents. Els algorismes executats per les GPU solen ser algorismes d'una gran intensitat aritmètica, algorismes que requereixen molts càlculs i tots sobre les mateixes dades,

per tant no necessiten gaires accessos a memòria i això estalvia ús de la caché envers d'un accés a memòria calculat, el que fa que aquest sigui molt més ràpid.

En un dispositiu CUDA existeixen diferents tipus de memòries, a la figura 5.4 es pot veure un petit esquema de com estan organitzades aquestes zones de memòria dins el hardware CUDA. Però, aquests tipus de memòria tenen objectius diferents, i són els següents:

Memòria global És la memòria per defecte. Si al codi no s'indica el contrari tota variable declarada només com a $\_\_device\_\_$ i les dades que s'envien des de la memòria principal del sistema cap a la GPU s'emmagatzemen en aquesta memòria. Aquesta memòria es disponible per a tots els fils d'execució llançats a la GPU i és la més lenta de totes.

Memòria compartida A cada bloc se li assigna una regió de memòria a la qual només els fils d'execució que hi hagin dintre d'aquest bloc hi poden accedir, així cada bloc té la seva memòria compartida. Aquesta memòria és més ràpida que la memòria global i no és accessible per a la CPU.

Memòria constant La memòria constant es troba al mateix nivell que la memòria global, accessible tant per a la CPU com per a la GPU. S'usa per emmagatzemar els valors constants que l'algorisme dels kernels han d'usar i és local per a cada multiprocessador de la GPU.

Memòria de Textures Aquest tipus de memòria és especial, ja que és accedida per un hardware concret diferent a la resta i s'ha de administrar d'una manera especial. Per tant, no es tindrà en compte el seu ús.

Registres És on s'emmagatzemen les variables locals del kernel. Cada fil d'execució té un espai limitat on s'emmagatzema la seva pròpia còpia de les variables locals del kernel, per això aquesta memòria és la més ràpida de totes. Com està lligada al cicle de vida del fil d'execució, les dades d'aquesta memòria es perden una vegada finalitza el fil d'execució.

Memòria local Es reserva per a tot el que no es pugui emmagatzemar en els registres quan aquests ja estan plens. A l'estar en el mateix nivell que la memòria global, la memòria local és lenta respecte als registres.

En una primera implementació només s'usarà la memòria global per defecte, així que com a treball posterior a aquest projecte es pot estudiar l'ús dels altres dos tipus de memòria per a intentar optimitzar el rendiment en l'execució de l'algorisme a la GPU.

5.2 El llenguatge CUDA C/C++

5.2.1 El model de programació paral·lela

El desenvolupament de dispositius multi-nucli en els darrers anys ha canviat el concepte envers el desenvolupament d'aplicacions. Fins a l'aparició de les primeres CPUs multi-nucli, el desenvolupador escrivia un algorisme amb el plantejament que aquest programa s'executaria sobre un sistema amb un únic fil d'execució. En canvi, ara, aquests sistemes han passat a ser sistemes multi-nucli, és a dir, sistemes multi-fil.

En aquests sistemes multi-fils, el desenvolupador té l'opció de crear una aplicació que aprofiti el potencial dels recursos hardware per millorar el rendiment del seu programa. Si bé, això es opcional i a més aquests sistemes són retrocompatibles, és a dir, un programa desenvolupat per a un sistema amb un sol fil d'execució continuarà funcionant en un de multi-nucli, encara que només aprofitarà un nucli de tots els que el hardware disposi. A més a més, aquest

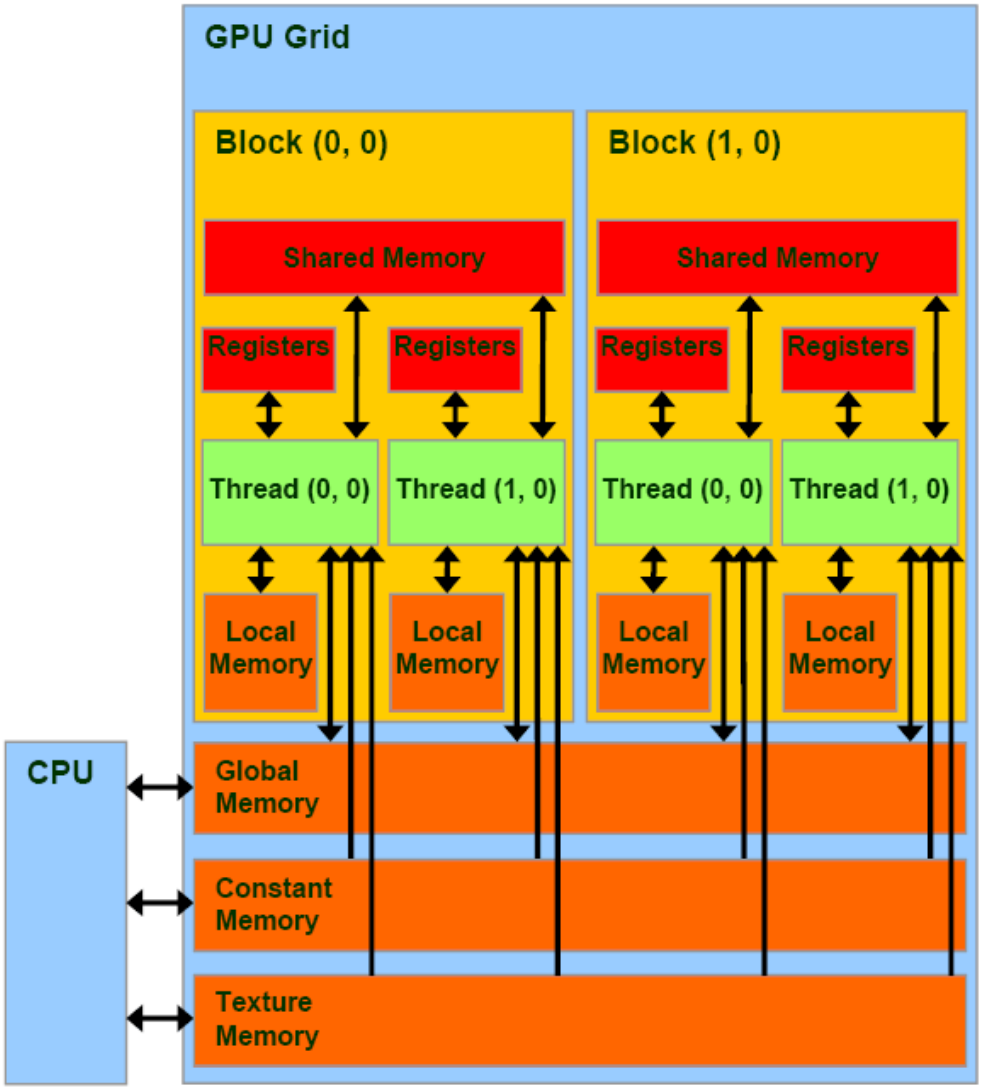

Figura 5.4: Arquitectura de les diferents memòries de la GPU[16]

entorn hardware de diferents nuclis d'execució ha de fer transparent per al desenvolupador el nombre d'unitats de processament disponibles al sistema. En altres paraules, un programa desenvolupat per a una plataforma multi-nucli ha de ser executable en diferents sistemes (de la mateixa arquitectura, això sí) amb diferent nombre d'unitats de procés. El problema és com fer que un desenvolupador pugui crear una aplicació per a un sistema hardware amb un cert nombre d'unitats de procés i que aquest software sigui totalment compatible amb dispositius multi-nucli futurs que tinguin moltes més unitats de processament. S'ha de tenir en compte la llei de Moore[4], que ens diu que cada any i mig es duplica el nombre de transistors dels circuits electrònics.

El model de programació paral·lela CUDA està pensat per a superar aquest repte, i a més per a que sigui fàcil d'aprendre per a aquells desenvolupadors que ja coneguin o dominin llenguatges d'alt nivell com C o C++. L'entorn CUDA es basa en els tres conceptes següents:

- Jerarquia de grups de fils d'execució.
- Accés a la memòria de la GPU.
- Sincronització dels fils.

Aquests conceptes permeten al desenvolupador poder dividir el seu problema en subproblemes més petits que poden ser resolts en blocs independents, i dintre d'aquest, en trossos més petits

on cada fil d'execució tracta una part del bloc. És una aplicació més de la màxima: "divideix i vèncer".

A la figura 5.5 podem veure un petit esquema d'aquesta idea. El desenvolupador pot decidir en quants blocs divideix el problema, i en temps d'execució el sistema sap com organitzar tots aquests blocs segons les unitats de procés disponibles al hardware. A la imatge es pot veure que per a una mateixa quantitat de blocs, segons els nuclis disponibles a la GPU, aquests s'organitzen automàticament.

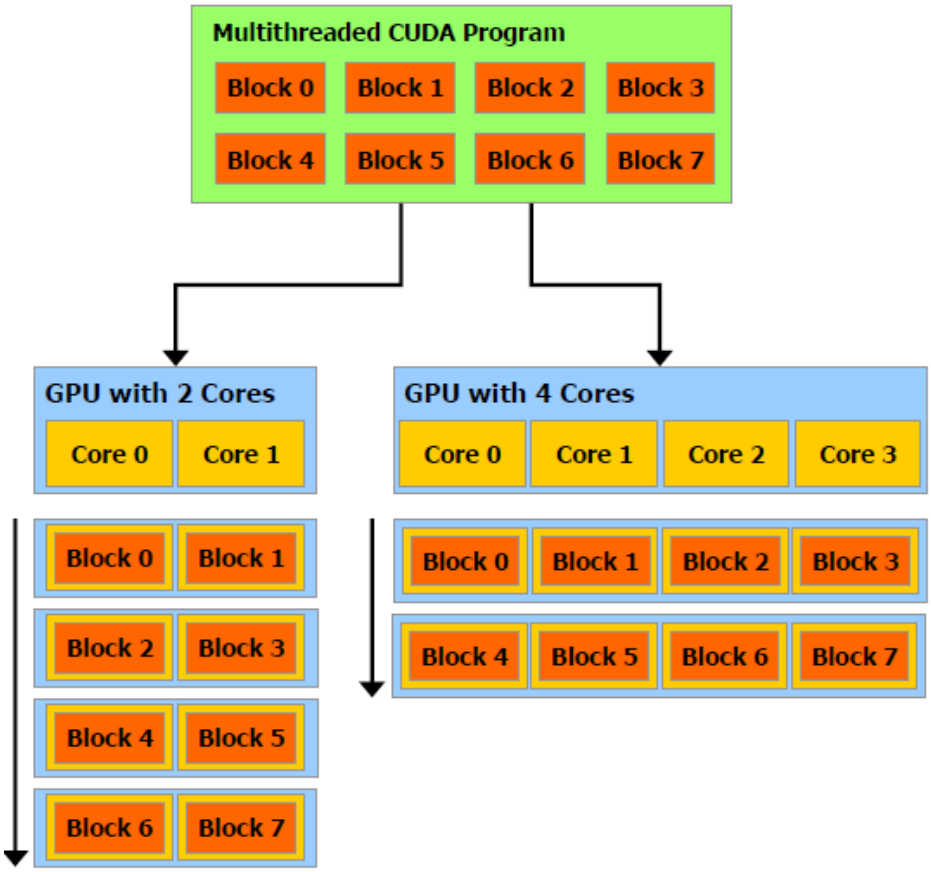

Figura 5.5: Execució d'un algorisme CUDA en GPU amb diferent nombre de nuclis[16]

Una altra característica que cal tenir en compte és la Capacitat de Càlcul o *Compute Capability*, que indica la versió de l'arquitectura del hardware i quines funcionalitats dintre la plataforma CUDA suporta. Aquesta propietat està definida per un nombre que indica la revisió major (versió de l'arquitectura del nucli) i un altre que indica la revisió menor (millora incremental de l'arquitectura). Així doncs, el hardware CUDA amb una capacitat de càlcul 1.X, on 1 indica el nombre de revisió major, és per a tot el hardware anterior a l'arquitectura anomenada Fermi, que correspon a la versió 2.x. La diferència entre aquestes dues arquitectures és la disponibilitat de diferents característiques com, per exemple, la definició d'una graella tridimensional, que a les versions menors a la 1.2 no es troba disponible. En el cas que ens ocupa, la Capacitat de Càlcul del hardware és 2.1.

5.2.2 Els kernels

Una vegada explicada la arquitectura del funcionament de CUDA, tot seguit s'explica com es pot desenvolupar un aplicatiu que s'executi sobre aquesta plataforma.

Dintre de la plataforma CUDA, les funcions executades per la GPU reben el nom de *kernel*, un kernel és la funció que s'executa N vegades en paral·lel per N fils d'execució dintre la GPU. Al contrari que les funcions de C en un sistema mono-nucli, que només s'executen en un sol fil.

La definició d'un kernel a CUDA C/C++ no varia gaire envers la declaració d'una funció en C/C++. En CUDA C/C++ un kernel es declara de la forma que es mostra a l'algorisme 5.1.

Algorisme 5.1 Prototipus de la declaració d'un kernel CUDA

```
__global__ void kernel(param1, param2, ...){
    /*Codi a executar pel kernel CUDA*/
}
```

On:

`__global__` Indica al compilador que es tracta d'un kernel CUDA. Les funcions declarades amb `__global__` són funcions que només es poden cridar des de funcions executades a la CPU. Existeixen més tipus d'etiquetes dintre de CUDA, però s'expliquen a la secció 5.2.3.

`void` A C, indica el tipus de dada de retorn de la funció, i el tipus `void` indica que la funció no retorna cap dada. A CUDA C/C++ els kernels no poden retornar cap dada. Si tenim N fils d'execució, no podem esperar que tots retornin alguna cosa, ja que si no, tindríem N valors retornats i només en podem guardar un. Llavors, com es poden obtenir les dades processades pel kernel?, això s'explica més endavant a la secció 5.2.4.

`kernel` És el nom que rep el kernel i amb el qual el cridem. Segueix les mateixes regles que els noms de les funcions a C/C++.

`(param1, param2, ..., paramN)` Són els paràmetres del kernel, segueixen les mateixes regles que els paràmetres de les funcions de C/C++.

Ara que ja s'ha explicat com es declara un kernel, a continuació s'exposa com es fa la seva crida, la seva execució dintre el codi del programa. Aquesta crida es fa tal i com es mostra a l'algorisme 5.2.

`Algorisme 5.2 Prototipus de la crida a un kernel CUDA`

```
kernel<<<N, M>>>(param1, param2, ...);
```

Tot seguit es descriu la sintaxi de l'algorisme 5.2:

`kernel` És el nom del kernel que volem executar. Ha de estar definit abans de la seva crida, d'altra manera, el compilador retornarà un error.

`<<<N, M >>>` És la configuració de la crida al kernel CUDA. Aquí s'especifica de quina forma (blocs i fils) s'executarà el kernel.

`(param1, param2, ..., paramZ)` Llista de paràmetres del kernel CUDA.

Cal destacar que existeix un límit temporal per a la crida al kernel, aquest límit es variable segons el sistema operatiu sobre el qual s'executa l'aplicació. Si l'execució del kernel triga massa, el programa retornarà els següent error: `the launch timed out and was terminated`. Aquest cas es pot donar si, per exemple, dintre del kernel s'executa un bucle infinit o l'algorisme es massa complex i necessita molt temps per a completar-se.

5.2.3 Tipus de funcions

Anteriorment s'ha vist que a la declaració d'un kernel, la primera paraula indica que aquesta funció s'executa a la GPU, i que llavors, es tracta d'un kernel CUDA. Però, dintre de CUDA es poden definir tres tipus de funcions, aquestes són les següents:

`__global__` Les funcions que es declarin amb aquest identificador són funcions que s'executen a la GPU i només les pot cridar una funció que s'executa a la CPU. Una funció que s'executa ala CPU sempre cridarà a un kernel d'aquest tipus.

`__device__` Les funcions que es declarin amb aquest identificador són funcions que s'executen a la GPU i només les pot cridar una altra funció que s'executi a la GPU. Es a dir, aquests tipus de funcions són kernels que només poden ser executats per un altre kernel, ja sigui `__global__` o `__device__`.

`__host__` Les funcions que es declarin amb aquest identificador són funcions que s'executen a la CPU, i que només poden ser cridades des d'altres funcions que s'executin a la CPU. Cal dir, que declarar una funció sense cap dels tres identificadors té el mateix efecte que declarar-la amb l'identificador `__host__`. És a dir, una funció sense cap d'aquests identificadors, s'executarà a la CPU.

5.2.4 Intercanvi de dades

Ara que ja s'ha explicat com es declara un kernel CUDA, se sap que aquests no poden retornar cap valor. Llavors, com podem recuperar les dades ja processades que es troben al dispositiu?

Quan es crida a un kernel se li passen una serie de paràmetres, són les dades que aquest necessita per poder executar l'algorisme programat. A la definició d'un kernel s'indica el número, tipus i nom de cadascun dels paràmetres que el kernel rebrà quan sigui cridat. Arribat a aquest punt cal fer una clarificació molt important. El paràmetres que rep un kernel CUDA poden ser paràmetres passats per còpia o per referència. Igual que com es fa a C/C++.

Paràmetre per valor A la funció cridada se li passa el valor d'una variable o un offset. Dintre de l'execució de la funció, aquest valor es una còpia de l'original que s'ha usat a la crida. Si aquest valor es modifica dintre la funció, una vegada acabada la seva execució aquest canvi no tindrà cap efecte sobre el valor original. Per exemple, una funció que rep un paràmetre per còpia de tipus enter es declara com es mostra a l'algorisme 5.3. Una crida a aquesta funció seria: `funció1(3)`, o bé `funció1(var)`. Essent `var` una variable de tipus enter amb un valor vàlid.

Paràmetre per referència A la crida de la funció aquest paràmetre és la direcció de memòria on es guarda el valor, és el valor real, no una còpia com a l'anterior. D'aquesta manera, tant la funció que fa la crida com la cridada veuen el mateix valor. Si la funció cridada modifica el valor de la posició, aquesta modificació afecta a la funció que ha fet la crida. Aquest tipus de paràmetres es declaren com a un punter (una direcció de memòria) a un tipus de dada, per exemple una funció que rep per paràmetre una referència a un tipus enter s'implementa com es pot veure a l'algorisme 5.4. Una crida a la funció de l'algorisme 5.4 seria: `funció1(&var)`, on `var` és una variable de tipus enter amb un valor vàlid. Cal dir que l'operador `&` que precedeix el nom de la variable indica que estem passant la direcció de memòria on s'emmagatzema aquesta variable.

Als kernels CUDA, ja que no poden retornar cap tipus de valor, s'usen el paràmetres per referència per fer arribar al programa que s'executa a la CPU les dades processades. Així

doncs, l'entrada de dades a un kernels serà un conjunt de paràmetres per còpia, i la sortida serà un conjunt de paràmetres per referència.

Ara que ja s'ha explicat com fer que un kernel tingui una entrada i una sortida de dades, falta un pas molt important. Si els kernels CUDA s'executen a la GPU, les dades han de residir a la memòria RAM de la pròpia GPU. Llavors, com s'ha de fer per a que les dades d'entrada estiguin a la RAM de la GPU en el moment que es cridi al kernel?

Algorisme 5.3 Declaració d'una funció amb un paràmetre per còpia

```
funció1(int param1){
    /*Codi que executa la funció*/
}
```

Algorisme 5.4 Declaració d'una funció amb un paràmetre per referència

```
funció1(int* param1){
    /*Codi que executa la funció*/
}
```

5.2.5 Enviant les dades a la GPU

Imaginem que donat un conjunt de N dades, un conjunt de nombres enters, han de ser processats per un algorisme que s'executa en un kernel CUDA a la GPU. Aquest conjunt està implementat en un vector declarat com es veu a l'algorisme 5.5.

Algorisme 5.5 Declaració del vector a la memòria principal

```
int Vector[N];
```

Abans de fer la crida al kernel aquestes dades han de ser transferides a la memòria de la GPU. Això es fa d'una manera transparent per al desenvolupador mitjançant unes crides a unes funcions CUDA ja implementades per a aquest propòsit. Encara que sembli estrany, s'ha de fer servir aquestes funcions encara que es podria pensar que l'enviament directament des de el disc dur de les dades cap a la GPU seria més eficient, una cosa semblant al concepte de la tecnologia DMA.

A C, per fer una còpia d'aquest buffer, es faria ús de les funcions pròpies de C `malloc()` i `memcpy()`. La primera per a reservar l'espai necessari per al nou vector i la segona per a copiar el contingut. Aquestes crides quedarien implementades com es veu a l'algorisme 5.6.

A la primera crida es reserva un espai de memòria de la mida de N enters. Cal dir, que la mida d'un enter s'obté amb la crida a la funció `sizeof()` pròpia de C, que donat un tipus de dada retorna la seva mida en bytes. `malloc()` retorna la direcció de memòria al primer byte de l'espai reservat, i aquesta direcció es guarda a `nouVector`, que prèviament s'ha declarat com a `int*`, un punter a una direcció de memòria on s'emmagatzema un tipus de dada enter. Cal aclarir que a C, el nom d'un vector indica la primera posició de memòria que aquest ocupa.

Algorisme 5.6 Reserva i còpia de memòria a C

```
nouVector = malloc(N*sizeof(int));
memcpy(nouVector, Vector, N*sizeof(int));
```

Amb la segona instrucció, copiem les dades apuntades per `Vector` a `nouVector`, l'espai de memòria que s'ha reservat amb la instrucció anterior. El tercer paràmetre de la funció `memcpy()` indica la quantitat de dades que s'han de copiar (en bytes), en aquest cas és el nombre d'elements enters multiplicat per la mida d'un enter en Bytes.

A l'entorn CUDA es segueix el mateix patró, encara que amb uns petits detalls propis. Igual que a C primer s'ha de reservar espai a la memòria de la GPU, per això es fa servir la funció `cudaMalloc()`. Aquesta funció té el mateix comportament que la funció `malloc()` pròpia de C però amb la memòria RAM de la GPU. Donat un punter i la mida de les dades on s'apunta, aquesta funció reserva l'espai indicat a la RAM de la GPU, en altre cas retorna un error. La reserva de memòria a la GPU per a l'exemple es fa amb la crida que es pot veure a l'algorisme 5.7.

Algorisme 5.7 Reserva de memòria a la GPU

```
error = cudaMalloc((void **) &Vector_gpu, N * sizeof(int));
```

En el exemple usat, `Vector_gpu` és un punter que indica la direcció en memòria de la GPU del primer byte reservat. A la crida a `cudaMalloc()` el primer paràmetre sempre ha de ser `(void **)&` tot seguit del nom del punter, i el segon la mida en bytes de l'espai a reservar. Aquesta funció retornarà 0 a la variable `error` si no hi ha hagut cap problema, i un valor diferent en qualsevol altre cas.

Una vegada tenim l'espai suficient reservat a la memòria de la GPU hem de copiar les dades. Això es porta a terme mitjançant la funció `cudaMemcpy()`. Per a l'exemple exposat, l'ús d'aquesta funció es mostra a l'algorisme 5.8.

Algorisme 5.8 Còpia de les dades des de la memòria principal cap a la GPU

```
error = cudaMemcpy(Vector_gpu,
                   Vector,
                   N * sizeof(int),
                   cudaMemcpyHostToDevice);
```

La crida es molt semblant a la funció `memcpy()` de C. El primer paràmetre és el punter a la posició de memòria destí de la còpia, el segon és un punter a l'origen de les dades, i el tercer és la mida de les dades a copiar. La gran diferència es troba a l'últim paràmetre, que és una constant que indica la direcció de la còpia de les dades.

A la gestió de la memòria de la GPU amb CUDA C/C++ es troben tres direccions diferents envers la còpia de dades:

`cudaMemcpyHostToDevice` Aquesta constant indica que s'estan copiant dades des de la memòria principal de la CPU cap a la memòria RAM de la GPU.

`cudaMemcpyDeviceToHost` Aquesta constant indica el contrari a l'anterior. S'utilitza per a copiar les dades ja processades des de la RAM de la GPU cap a la RAM principal de la CPU.

`cudaMemcpyDeviceToDevice` Aquesta constant s'utilitza per a copiar dades dintre la mateixa RAM de la GPU.

Després de tot això, a la RAM de la GPU estan copiades les dades de `Vector` a l'espai apuntat per `Vector_gpu`.

Cal remarcar, però, que la gestió de la memòria de la GPU sempre es fa des de un algorisme executat per la CPU. Els kernels no poden realitzar aquesta gestió. Internament, tant `cudaMalloc()` com `cudaMemcpy()` són kernels declarats `__global__` per tant, si intentem compilar un programa amb un kernel que tingui una d'aquestes instruccions, el compilador retornarà un error.

A l'igual que es reserva memòria a la GPU, al finalitzar el programa cal alliberar-la. Això es duu a terme amb la funció:

Algorisme 5.9 Alliberament de la memòria reservada a la GPU

```
cudaFree(Vector_gpu);
```

Després cal, també alliberar la memòria assignada al vector original amb la funció de C `free(Vector)`.

Ja s'ha explicat com es poden intercanviar dades entre la memòria principal de l'ordinador i la memòria de la GPU, encara que aquest algorisme d'intercanvi de dades només s'usa si la mida de les dades és variable i s'han d'intercanviar amb la memòria principal. Però si, per exemple, l'algorisme que ha d'executar la GPU necessita d'una taula o vector que només s'usarà per a la seva consulta es pot declarar aquest element directament a la memòria de la GPU tal i com es mostra a l'algorisme 5.10, on es declara i inicialitza un vector de deu posicions.

Algorisme 5.10 Declaració d'un vector directament a la memòria de la GPU

```
__device__ const vector[10] = {0,1,2,3,4,5,6,7,8,9};
```

5.2.6 Cridant al kernel

Una vegada tenim les dades necessàries copiades a un espai de la memòria de la GPU. Ja es pot cridar al kernel per a que les tracti. Si cal, també s'haurà reservar espai a la memòria de la GPU per les dades resultants fent servir la funció `cudaMalloc()`.

Ara es moment de cridant al kernel CUDA. En el nostre exemple, el kernel que incrementa està declarat com es veu a l'algorisme 5.11.

Algorisme 5.11 Declaració del kernel `incrementa()`

```
__global__ void incrementa(int* vector){
    /*Codi a executar*/
}
```

Llavors, per cridar al kernel incrementa es fa tal i com mostra l'algorisme 5.12.

La crida a un kernel CUDA és semblant a la crida d'una funció C. La crida comença amb el nom del kernel seguit de la configuració de la crida i dels paràmetres del kernel. A l'exemple usat, el kernel `incrementa()` només rep un paràmetre, un punter a les dades que es troben a la memòria de la GPU.

Algorisme 5.12 Crida al kernel incrementa()

```
incrementa<<<1, N>>>(Vector_gpu);
```

5.2.6.1 La configuració de la crida

A l'exemple usat, la crida al kernel s'ha fet amb la configuració de l'algorisme 5.13.

Algorisme 5.13 Configuració de la crida al kernel incrementa()

```
<<<1, N>>>
```

Aquesta es la configuració de la crida al kernel CUDA. Aquests dos paràmetres indiquen les característiques del paral·lelisme amb el qual executarà el kernel. Aquests dos paràmetres indiquen el nombre de blocs (*1* en aquest cas) i el nombre de fils d'execució que tindrà cada bloc (*N* en aquest cas).

La plataforma CUDA permet que aquests dos paràmetres siguin de dos tipus: enters (tipus `int`) o `dim3`. El tipus de dada `dim3` es un tipus propi de CUDA, aquest pot representar una graella (*grid*) de blocs, o bé fils, que pot ser unidimensional, bidimensional o tridimensional. Cal tenir en compte, que aquells paràmetres a la declaració que no tinguin valor, per defecte valdran 1. La declaració d'una variable d'aquest tipus és com es mostra a la figura 5.2

Algorisme 5.14 Prototipus de la declaració d'una variable de tipus *dim3*

```
dim3 variable(X, Y, Z);
```

Un aspecte important a l'hora de configurar la crida a un kernel CUDA, es tenir en compte l'organització dels blocs i dels fils d'execució. Per exemple, si volem processar una imatge, podem assignar un bloc a cada píxel d'aquesta, i en aquest cas podem mapejar cada píxel de la imatge a tractar a un bloc. Per exemple, si declarem una variable `dim3` per a la graella de blocs i per als fils dintre de cada bloc de la següent manera com mostra l'algorisme 5.15.

El resultat serà una configuració amb una graella de blocs de 3x2, i dintre de cada bloc es crearan 4x3 fils d'execució. A la figura 5.6 es pot veure gràficament aquest exemple.

Cal destacar, però, que els dos paràmetres de la configuració del kernel tenen un límit definit pel hardware on s'executa la aplicació CUDA. Per exemple, si la versió de la capacitat de càlcul del hardware CUDA no és 2.0 o superior, no es poden definir graelles de blocs tridimensionals. Aquests paràmetres es poden obtenir executant un dels exemples que s'inclou dintre del SDK de CUDA. Es tracta del programa *deviceQuery.exe*, que pel hardware usat per a aquest projecte retorna la informació que podem veure a l'apartat 2.4.1.

En el cas del hardware emprat en aquest projecte, la mida màxima de la graella de blocs és de 65535 * 65535* 65535 blocs i, a la vegada, dintre de cada bloc es poden executar fins a un total de 1024 fils d'execució.

5.2.7 Identificació del fil

Una vegada s'ha fet la crida al kernel tenim, seguint el mateix exemple anterior, *1*N* fils d'execució a la GPU que segueixen el mateix codi, el codi implementat al kernel `incrementa()`. La nostra idea de paral·lelisme ensopega, en aquest moment, amb un altre obstacle. Es vol que cada fil d'execució de la GPU tracti una, i només una, dada del vector que s'ha transferit a la

Algorisme 5.15 Declaració dels blocs i els fils amb variables `dim3`

```
dim3 grid(3, 2);
dim3 threads(4,3);
```

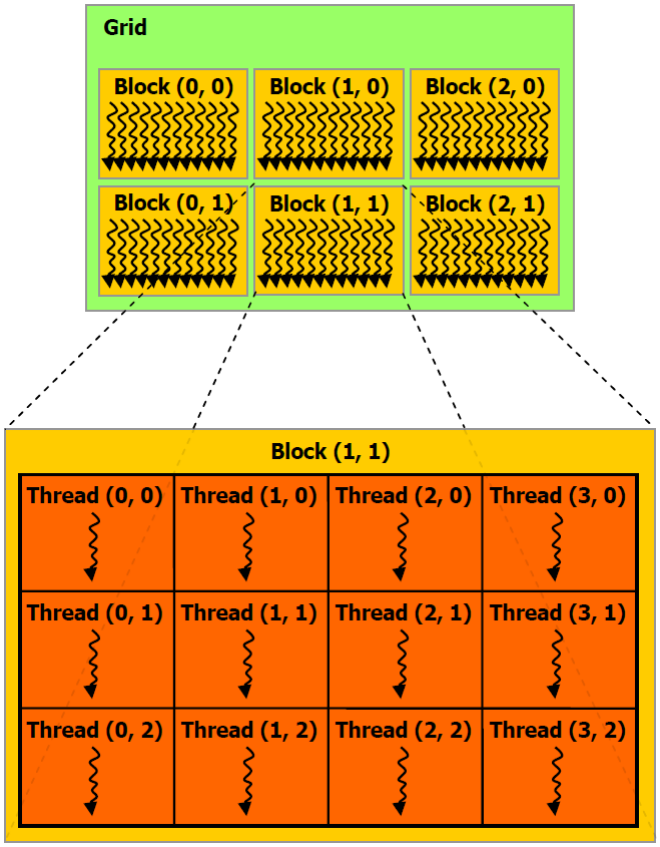

Figura 5.6: Esquema de l'organització de blocs i fils d'execució generats a la GPU[16]

memòria RAM de la GPU. Però, si tots els kernels executen el mateix codi, com es distingeix cada fil?

Dintre del kernel es poden usar una sèrie de variables existents dintre de CUDA. Aquestes variables són:

- Identificació del fil, posició X, Y i Z dintre del bloc. Les variables que ho indiquen són: `threadIdx.x`, `threadIdx.y` i `threadIdx.z`.
- Identificació del bloc, posició X, Y i Z dintre de la graella. Les variables que ho indiquen són: `blockIdx.x`, `blockIdx.y` i `blockIdx.z`.
- Dimensions del bloc. Les variables que ho indiquen són: `blockDim.x`, `blockDim.y` i `blockDim.z`.
- Dimensions de la graella. Les variables que ho indiquen són: `gridDim.x`, `gridDim.y` i `gridDim.z`.

En el nostre exemple, donat que la crida al kernel s'ha configurat amb un sol bloc i N fils per cada bloc, amb la variable `threadIdx.x` és suficient per a mapejar cada fil a una posició del vector de dades amb un accés directe al vector de la forma que es pot veure a l'algorisme 5.16.

Això es pot fer ja que tots els fils es troben dintre un mateix bloc i les dades són un vector unidimensional, però què passaria si la crida es fes amb la configuració de blocs i fils que mostra l'algorisme 5.17?

Algorisme 5.16 Accés directe a la dada corresponent del vector

```
vector_gpu[threadIdx.x]++;
```

Algorisme 5.17 Crida amb una configuració de dos blocs

```
incrementa <<<2, N>>>(vector_gpu);
```

El nombre total de fils és el doble que al cas anterior, llavors amb la simple consulta a `threadIdx.x` no és suficient. Com que aquesta variable és local a cada bloc, a la crida de la nova configuració existeixen dos fils amb el mateix valor per a la variable threadIdx.x!. Doncs, s'ha de re-formular la identificació del fil.

Per generalitzar la identificació del fil de forma absoluta, és a dir en quin fil s'executa el codi dintre de tots els fils que es generen a tots els blocs, al començament de la implementació de cada kernel es declara una variable de tipus enter on s'emmagatzema l'identificador del fil que executa el codi del kernel, el que es coneix amb el nom d'*offset*. Aquest offset es genera de la mateixa forma en la que en C volem accedir a una posició d'una matriu de forma calculada. Pel cas de la nova configuració, l'*offset* s'haurà de calcular tal i com es mostra a l'algorisme 5.18.

Algorisme 5.18 Identificació del fil amb la nova configuració

```
int thread_idx = blockIdx.x * threadIdx.x;
```

El càlcul de l'offset depèn de quina és la configuració de la crida al kernel, així, com més complexa sigui aquesta configuració, més complex serà el càlcul de l'identificador de l'offset. En el cas que la graella sigui tridimensional i que els fils d'execució dintre un bloc estiguin organitzats en una altra graella bidimensional, cal tenir en compte les dimensions de la graella de blocs, les dimensions de cada bloc, i el nombre fils de cada bloc.

Ara bé, amb la nova configuració, hi ha més fils que dades a processar, que es fa dels fils que no han de fer res? La solució a aquest problema és bastant simple, només cal passar per paràmetre al kernel nombre de fils que s'han llençat i després de calcular l'*offset* s'utilitza el condicional ficant-hi a dins el codi del kernel que es mostra al codi 5.19.

Així, si el fil actual té un identificador superior al nombre necessari de fils, no executa el codi del kernel. Això comporta un altre problema, si hi ha fils que no fan res, aquests acaben abans que els que tenen codi a executar. En aquest cas, s'utilitza la funció interna de CUDA `__syncthreads()`. L'objectiu d'aquesta funció és molt senzilla, el fil d'execució es queda parat en aquesta instrucció fins que tota la resta dels fils arribin a la mateixa línia. La sol·lució al problema dels fils desocupats es posar la instrucció de l'algorisme 5.20 just després del condicional anterior.

5.2.8 L'exemple sencer

Ara que ja s'han explicat totes les parts necessàries, l'exemple usat tindria un codi semblant al que es pot veure a l'algorisme 5.21.

Algorisme 5.19 Control per a fils generats que no tenen dades disponibles

```
if(thread_idx < N){
    /*Algorisme del kernel*/
}
```

Algorisme 5.20 Sincronització dels fils de la GPU

```
if(thread_idx < N){
    /*Algorisme del kernel*/
}
__syncthreads();
```

Algorisme 5.21 Codi complert de l'exemple usat

```
/*Nombre de dades al vector*/
#define N 10
__global__ void incrementa(int* vector, int nthreads){
    /*Identificació del fil a la GPU*/
    int thread_idx = threadIdx.x;
    /*Si no hi ha dades suficients per a aquest fil, aquest no fa res*/
    if(thread_idx < nthreads){
        vector[thread_idx]++;
    }
    /*Espera a que tots els fils acabin la seva execució*/
    __syncthreads();
}
int main(){
    int Vector[N];
    int *Vector_gpu;
    /*Inicialització del vector a la RAM principal*/
    for(int i = 0; i < N; i++) Vector[i] = i;
    /*Reserva d'espai i còpia de les dades a la GPU*/
    cudaMalloc((void**)&Vector_gpu, N*sizeof(int));
    cudaMemcpy(Vector_gpu, Vector, N*sizeof(int), cudaMemcpyHostToDevice);
    /*Crida al kernel*/
    incrementa<<<1, N>>>(Vector_gpu, N);
    /*Recuperació de les dades processades*/
    cudaMemcpy(Vector, Vector_gpu, N*sizeof(int), cudaMemcpyDeviceToHost);
    /*Alliberació de la memòria reservada*/
    cudaFree(Vector_gpu);
    free(Vector);
    return 0;
}
```

Capítol 6

Libvpx i AES

6.1 La biblioteca libvpx

Com ja s'ha explicat a la descripció del projecte, un dels exemples usat per a analitzar la viabilitat de la tecnologia CUDA és la biblioteca de vídeo libvpx. De totes les funcionalitats que disposa la biblioteca, aquest projecte es centra en la descodificació dels fotogrames. Cal dir també, que aquesta biblioteca no tracta el so, només el vídeo, per a tractar el so s'haurà d'usar una biblioteca expressament implementada per a aquesta tasca. Però abans d'entrar en el funcionament de l'algorisme cal aclarir certs conceptes previs sobre la gestió dels fotogrames.

Per sintetitzar el contingut d'aquest capítol, i ja que no aporta cap valor afegit, s'ha omès parlar sobre quin tipus de format usa aquesta biblioteca i com aquesta organitza les dades dintre de l'arxiu.

6.1.1 La preparació el codi font

El codi font de la biblioteca es troba disponible per a la seva descarrega en un arxiu comprimit, però una vegada s'han extret tots els fitxers s'ha de configurar el codi font per a generar els programes d'exemple. Aquesta configuració la fa un executable que ja està inclòs al paquet descarregat, l'únic que cal tenir en compte és la configuració que cal indicar-li a aquest programa per a que generi el codi font adaptat al sistema on es troba. Si aquest programa s'executa sense cap paràmetre, automàticament detecta el tipus de CPU present al sistema i tots els seus conjunts d'instruccions en llenguatge assemblador que pot interpretar. D'aquesta manera el codi resultant conté certes parts escrites en llenguatge assemblador per tal d'optimitzar l'execució de la biblioteca en la CPU detectada, i que no es poden traduir a CUDA C.

Per a poder generar una versió completament en C del codi font de la biblioteca cal dir-li expressament al programa configurador executant-lo amb el paràmetre `--target=generic-gnu`. Això configurarà la biblioteca per a no utilitzar cap optimització, després només cal compilar els exemples amb la comanda `make` i es podrà començar el desenvolupament de la versió CUDA. tot el procés de la configuració del codi font es mostra a l'algorisme 6.1. Cal tenir en compte que el desenvolupament es fa en un entorn GNU/Linux i que cal donar-li permisos d'execució tant al programa configurador com a un script intern amb la comanda `chmod`.

Algorisme 6.1 Comandes per a generar una versió en C del codi font de libvpx

```
$>chmod +x configure
$>chmod +x /build/make/version.sh
$>./configure --target=generic-gnu
$>make
```

6.1.2 Els tipus de fotogrames

Des dels seus inicis, el cinema sempre s'ha basat en una successió d'imatges estàtiques lleugerament diferents que mostrades successivament amb una certa rapidesa dóna la sensació de moviment. En els formats informàtics actuals es segueix el mateix principi, un arxiu de vídeo no és més que una successió d'imatges que es mostren successivament a una velocitat suficient per a que l'ull de l'espectador pugui percebre el moviment juntament amb el so que acompanya a les imatges.

Cada biblioteca de compressió de vídeo tracta els fotogrames d'una forma diferent, encara que entre algunes biblioteques es mostra un tractament semblant. A la biblioteca libvpx els fotogrames es classifiquen en diferents tipus, aquests són:

Intraframe Aquests fotogrames són descomprimits sense tenir en compte cap altra fotograma. Aquest tipus de fotograma també rep el nom de Keyframe. El descodificador reinicia el seu estat si troba un fotograma d'aquest tipus. En un vídeo comprimit amb aquesta biblioteca, el seu primer fotograma serà sempre d'aquest tipus.

Interframe Aquests tipus de fotograma es descodifiquen tenint en compte l'últim Intraframe tractat (fent-hi referència) i tots els Interframes que es troben entre l'últim Intraframe i el fotograma actual. El que fa el descodificador es predir, basant-se en l'últim fotograma processat, el contingut del següent fotograma.

Gold Reference Frame El descodificador escull un fotograma de forma arbitrària d'entre els fotogrames anteriors a l'actual per a usar-lo en la descodificació, aquests fotogrames escollits de forma arbitrària són els Gold Reference Frames. Per exemple, aquests tipus de fotogrames s'usen per a mantenir un coneixement dels elements que apareixen al fons d'un fotograma.

Alternate Reference Frame Aquests tipus de fotogrames poden ser mostrats a la reproducció del contingut o no. L'objectiu d'aquest tipus de fotogrames és emmagatzemar dades útils sobre la compressió del vídeo per ajudar a una millor descompressió de la resta dels fotogrames.

Aquest projecte no té com a objectiu entendre el format ni el funcionament específic de la biblioteca libvpx així que, per a simplificar-ho, a partir d'aquest punt només es parlarà dels dos primers tipus de fotogrames: els Keyframes i els Interframes. D'aquesta manera, podem visualitzar el contingut d'un vídeo com una successió de Keyframes i Interframes tal i com es mostra a la figura 6.1.

Figura 6.1: Estructura d'un vídeo de n fotogrames generat per libvpx

6.1.3 La descodificació d'un fotograma

Com s'ha explicat a l'apartat anterior, els Interframes necessiten que les dades del Keyframe i dels Interframes anteriors al fotograma que s'està processant per a poder tractar el fotograma actual fins a trobar un nou fotograma clau. Doncs, la descompressió a nivell de fotogrames és seqüencial. El descodificador s'inicialitza en el primer fotograma que troba, un fotograma clau,

i a partir d'aquest va tractant tots els fotogrames del tipus Interframe que en troba fins que es troba amb un altre fotograma clau, llavors reinicia el seu estat.

Fins ara, s'ha fet una introducció d'alt nivell al tractament que en fa libvpx dels fotogrames d'un vídeo comprimit. Però baixant més el nivell d'abstracció, com tracta la biblioteca la descompressió de cada fotograma?

6.1.3.1 Els macroblocs

La biblioteca libvpx, a l'hora de descodificar, divideix cada fotograma en divisions de la mateixa mida, aquests elements s'anomenen macroblocs i contenen les dades d'un grup de píxels del fotograma que es tracta. Aquesta tècnica s'utilitza molt habitualment en la compressió/descompressió d'imatges. Per exemple, el format de compressió d'imatges JPEG . Per tant, podem visualitzar un fotograma com a una matriu de macroblocs, tal i com mostra la figura 6.2.

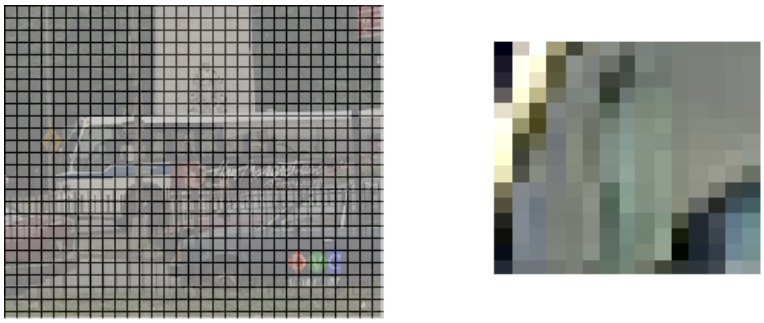

Figura 6.2: Divisió d'un fotograma en macroblocs (esq.) i contingut d'un macrobloc (dreta)

L'algorisme de libvpx tracta el vídeo al nivell d'aquestes unitats, així que per el propòsit d'aquest projecte no cal baixar més en el nivell d'abstracció. L'objectiu és implementar una versió CUDA de l'algorisme de la biblioteca sense haver d'analitzar-lo en la seva totalitat.

6.1.3.2 Les particions

Aquesta biblioteca està preparada per a permetre un cert nivell de paral·lelisme. Els seus autors ja han tingut en compte l'actual proliferació de processadors multi-nucli, i és per això que han implementat la biblioteca libvpx per a poder aprofitar aquest nou hardware. Ara bé, com s'implementa aquesta paral·lelització?

La idea és molt senzilla, es divideix les dades del fotograma en tants trossos com nuclis tingui la CPU, tal i com es mostra a la figura 6.3. Aquests trossos s'anomenen particions, i la biblioteca libvpx suporta fins a vuit particions d'un mateix fotograma. Així, cada partició té un conjunt de macroblocs consecutius que pertanyen al mateix fotograma, i la biblioteca processa cadascuna d'aquestes particions en un nucli de la CPU. Però, això no es fa de manera automàtica, és el compressor el que crea aquestes particions i fica el nombre d'aquestes en el que es divideix el fotograma a la capçalera d'aquest. Així que si es descomprimeix un vídeo amb aquesta biblioteca però el programa usat per a comprimir el vídeo no ha afegit aquesta informació, el descompressor no l'aprofitarà.

Aquesta previsió de poder dividir un fotograma en fins a vuit particions, no és pas dolenta, les CPU d'escriptori més potents existents al mercat actualment, són fabricades amb quatre nuclis físics i amb tecnologia Hyper Threading (en el cas de les CPU Intel), el que fa que puguin doblar el nombre de fils d'execució.

Per tant, el paral·lelisme que ofereix aquesta biblioteca es limita al nombre de fils d'execució possibles a la CPU on s'executi, i comparant aquest nombre de fils amb el nombre de fils d'execució possibles a una GPU amb CUDA es queda bastant curt.

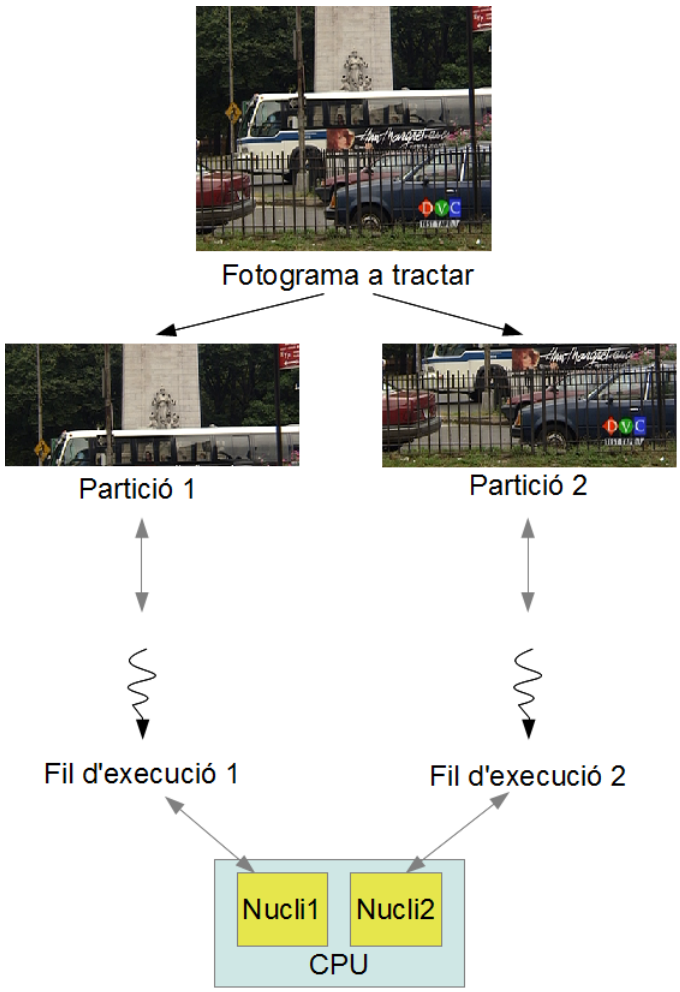

Figura 6.3: Acceleració del tractament d'un fotograma mitjançant particions en una CPU de 2 nuclis

6.1.4 La versió CUDA de la biblioteca

Per a fer el desenvolupament de la versió CUDA més senzill, s'han omès tots els fitxers del codi font de la biblioteca innecessaris. Aquests fitxers contenen funcionalitats de la biblioteca que el programa *simple_decoder* no usa, com per exemple diferents filtres que s'usen per el post-processat del vídeo (són efectes que s'afegeixen al vídeo una vegada descomprimit, en el cas de *simple_decoder*). A aquesta nova versió se l'ha anomenat *simple_decoder_lite*, per ser una versió amb menys funcionalitats que l'original. Una vegada aquesta versió reduïda de la biblioteca compila correctament, es procedeix a la seva modificació.

6.1.4.1 Diferents estratègies per paral·lelitzar libvpx

Ara que ja s'ha explicat el contingut necessari per a entendre, en un alt nivell, el funcionament de la biblioteca libvpx, a continuació s'explica què i com s'ha dut a terme la paral·lelització d'aquesta amb la tecnologia CUDA. A l'hora de fer aquesta paral·lelització, es plantegen les següents estratègies, i la seva viabilitat, per afrontar aquest problema i tot seguit s'explica perquè són (o no) viables):

- Paral·lelització a nivell de fotograma (No és viable)
- Paral·lització de grups de fotogrames (No és viable)
- Paral·lelització de particions (És viable)

- Paral·lelització a nivell de macroblocs (És viable)

Començant per la primera opció, la paral·lelització a nivell de fotograma, aquesta estratègia planteja una contradicció amb el funcionament de la biblioteca. Anteriorment s'ha vist que per a descomprimir un fotograma del tipus Interframe cal primer tenir les dades descomprimides dels fotogrames que el precedeixen. Això planteja un problema, si el tractament dels fotogrames entre un Keyframe i el següent és seqüencial degut a la predicció entre fotogrames, no es pot fer un paral·lelsime a nivell de fotograma. És a dir, no podem assignar un fil d'execució CUDA a cada fotograma per a que el processi, perquè en el cas dels Interframes hauria d'esperar a que els fils encarregats de generar les dades dels fotogrames anteriors finalitzessin la seva tasca per a començar a processar el fotograma assignat. Això desmunta el paral·lelisme que es vol assolir, per tant aquesta opció no és viable.

La següent opció és paral·lelització de grups de fotogrames. És a dir, basant-se en els motius que fan invàlida l'estratègia anterior, la seqüencialitat del tractament dels Interframes i tenint en compte que l'estat del descodificador es reinicia a cada Keyframe que es troba es pot deduir que a cada Keyframe el descodificador tindrà la mateixa informació. Aquesta estratègia planteja assignar el tractament d'un Keyframe i els fotogrames que depenen d'ell a un fil d'execució de la GPU. Però, es planteja una altra pregunta, cada quants fotogrames es troba un fotograma clau?, o formulada d'altra manera, cada quants fotogrames cal generar un fotograma clau?

A primera vista es podria pensar que els fotogrames clau no tenen sentit, és a dir, partint del primer fotograma no es poden tractar tota la resta de fotogrames de l'arxiu? Doncs no. Els fotogrames estan pensats per a fer més eficient la compressió de les dades del vídeo. Aquest tipus de fotogrames fan falta quan un fotograma actual és molt diferent al immediatament anterior, i això es produeix normalment quan al vídeo hi ha talls produïts per canvis en els plans de la imatge. Llavors, si l'aparició d'un fotograma clau depèn dels talls al vídeo, no es pot assegurar que n'apareguin en períodes regulars. Aquesta estratègia té un gran inconvenient. Si, per exemple, la mida de les dades que s'han de tractar és més gran que la mida de la memòria disponible a la GPU, simplement no es pot dur a terme aquesta estratègia. Per aquest motiu, tractar blocs de fotogrames queda descartat.

La tercera opció és possible d'implementar en CUDA de la mateixa manera (conceptualment parlant) que es fa a la CPU. S'assigna un fil d'execució de la GPU a cadascuna de les particions per a que en facin el seu tractament. Aquesta estratègia, l'únic que fa és agrupar els macroblocs del fotograma en particions. Llavors perquè no assignar, per exemple, un fil d'execució a cada fila de macroblocs?

Finalment, la quarta i última opció planteja el tractament a nivell de macrobloc. Una vegada que el descodificador està llest per a començar a descomprimir els macroblocs del fotograma, no li cal més informació i pot tractar-los de forma individual i independent. Així doncs, és més eficient pensar en assignar un fil d'execució de la GPU al tractament d'una fila de macroblocs, ja que aquesta tasca, a més a més, és totalment mecànica. Per tant, es decideix implementar aquesta última estratègia en la versió CUDA de la biblioteca libvpx.

A la implementació inicial s'assigna una fila a cada fil d'execució a cadascuna de les files de macroblocs que componen el fotograma. Però és fàcil arribar a la conclusió que seria més òptim, més eficient i a més acceleraria l'execució molt més, s'hi s'assignés un fil d'execució de la GPU al tractament de cada macrobloc del fotograma. Tot i així, per tal que el codi CUDA resultant sigui el més semblant possible al codi original, s'ha decidit no fer-ho i assignar els fils d'execució de la GPU a cada fila de macroblocs tal i com es mostra a la figura 6.4. Les possibles optimitzacions del codi, com la que s'acaba de comentar es deixen per a treball futur sobre el projecte.

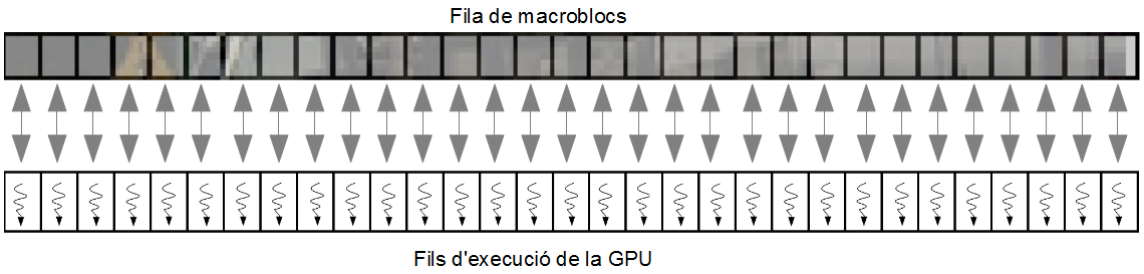

Figura 6.4: Assignació de fils d'execució de la GPU a una fila de macroblocs del fotograma

6.1.4.2 La descodificació per macroblocs de libvpx

La tecnologia CUDA està pensada per a paral·lelitzar algorismes mecànics que demanen de moltes operacions sobre les mateixes dades. Habitualment aquest tipus d'algorismes s'implementen amb bucles a les versions CPU, i són aquests bucles els que CUDA fa desaparèixer. En el cas que libvpx, bona part del cost temporal de l'algorisme es troba en les operacions per a transformar les dades comprimides als seus valors originals en el tractament dels macroblocs.

A la implementació original de la biblioteca, la descompressió del fotograma es duu a terme a la funció `vp8_decode_frame()` que es troba al fitxer *decodframe.c*. Aquesta funció entre d'altres coses, prepara el descodificador per a la descompressió del fotograma i fa la predicció del fotograma actual segons les dades que té del fotograma anterior. De fet, la descompressió del fotograma es realitza al final d'aquesta funció. El codi original de la biblioteca que descodifica el fotograma es pot veure a l'algorisme 6.2.

Algorisme 6.2 Recorregut de les files de macroblocs del fotograma

```
int vp8_decode_frame(VP8D_COMP *pbi){
    [...]
    /* Decode the individual macro block */
    for (mb_row = 0; mb_row < pc->mb_rows; mb_row++){
        if (num_part > 1){
            xd->current_bc = & pbi->mbc[ibc];
            ibc++;

        if (ibc == num_part)
            ibc = 0;
    }
    decode_mb_row(pbi, pc, mb_row, xd);
    }
}
```

Com es pot veure a l'algorisme, el descodificador itera per a cadascuna de les files de macroblocs en les que es divideix el fotograma. Dintre de la funció decode_mb_row() el descodificador itera per a cada columna de la fila i processa el macrobloc corresponent, tal i com es mostra a l'algorisme 6.3.

Abans de la crida la funció `decode_macroblock()`, qui finalment realitza la descompressió del macrobloc actual, s'actualitzen algunes variables relacionades amb el macrobloc que s'està tractant.

Així doncs, segons l'estratègia escollida, com s'assigna un fil d'execució de la GPU a la descompressió de cada filera de macroblocs del fotograma, no cal recòrrer el fotograma com una matriu de macroblocs, si no que es mapeja la macrobloc que es tractarà segons l'identificador

Algorisme 6.3 Recorregut de les columnes de macroblocs del fotograma

```
static void decode_mb_row(VP8D_COMP *pbi, VP8_COMMON *pc, int mb_row, MACROBLOCKD *
    [...]
    for (mb_col = 0; mb_col < pc->mb_cols; mb_col++){

        [...]
        decode_macroblock(pbi, xd, mb_row * pc->mb_cols  + mb_col);
        [...]
    }
}
```

del fil de la GPU.

Una vegada decidida l'estratègia i comprès com aplicar-la es passa a la implementació de la descodificació del fotograma amb CUDA.

6.1.4.3 Descomprimint un fotograma amb CUDA

Si s'examinen el arxius de codi font de la biblioteca libvpx es pot apreciar que és molt extens. L'objectiu és accelerar la biblioteca canviant únicament el codi font necessari, per aquest motiu s'ha decidit que la implementació de la descodificació del fotograma en CUDA es farà en una biblioteca estàtica. D'aquesta manera, el manteniment del codi es més fàcil, ja que el codi modificat queda encapsulat a la biblioteca i no es barreja amb el codi original que només depèn de la biblioteca estàtica. Per a que això sigui cert, es crea una funció embolcall que fa tota la gestió necessària amb la GPU, és a dir, la gestió de la transferència de les dades i la crida al kernel CUDA.

Però per a implementar una versió CUDA de la funció decode_mb_row() cal implementar totes les funcions que aquesta crida i els tipus de dades que siguin necessaris. Encara que el que es fa és paral·lelitzar la crida a aquesta funció, cal recordar que un kernel CUDA no pot fer ús d'una funció que no estigui definida a la GPU per tant, cal definir totes les funcions que hi han per sota de `decode_mb_row()` com a kernels CUDA que només s'executen a la GPU. Per tal de diferenciar les versions CUDA de les implementacions originals s'ha pensat en posar el prefix "CUDA" a tota funció i tipus de dada de libvpx del qual s'hagi de fer una versió CUDA. Aquesta és una solució simple i efectiva, ja que la identificació de les funcions originals és trivial.

Per tant, amb aquesta nova configuració, la nova crida a la funció `decode_mb_row()` queda com a una crida a una funció wrapper que serà publica a la biblioteca estàtica que s'ha creat, tal i com es mostra a l'algorisme 6.4.

Algorisme 6.4 Crida a la funció embolcall CUDA

```
int vp8_decode_frame(VP8D_COMP *pbi){
    [...]
    CUDA_decode_macroblock_rows(pbi, pc, xd);
    [...]
}
```

Dintre de la funció embolcall `CUDA_decode_macroblock_rows()` es gestiona la reserva de memòria de la GPU, la transferència de dades, la crida al kernel, la recollida de les dades processades i l'alliberament de la memòria reservada. Doncs, la implementació de la funció `CUDA_decode_macroblock_rows()` queda com es mostra a l'algorisme 6.5.

Algorisme 6.5 Implementació de la funció `CUDA_decode_macroblock_rows()`

```
void CUDA_decode_macroblock_rows(CUDA_VP8D_COMP *pbi, CUDA_VP8_COMMON *pc, CUDA_MAC
    CUDA_VP8D_COMP *gpu_pbi;
    CUDA_VP8_COMMON *gpu_pc;
    CUDA_MACROBLOCKD *gpu_xd;
    /*Reserva d'espai a la GPU pel paràmetre 'pbi', 'pc' i 'xd'*/
    cudaMalloc((void**) &gpu_pbi, sizeof (CUDA_VP8D_COMP));
    cudaMalloc((void**) &gpu_pc, sizeof (CUDA_VP8_COMMON));
    cudaMalloc((void**) &gpu_xd, sizeof (CUDA_MACROBLOCKD));
    /*Copia les dades dels tres paràmetres a la GPU*/
    cudaMemcpy(gpu_pbi, pbi, sizeof (CUDA_VP8D_COMP), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_pc, pc, sizeof (CUDA_VP8_COMMON), cudaMemcpyHostToDevice);
    cudaMemcpy(gpu_xd, xd, sizeof (CUDA_MACROBLOCKD), cudaMemcpyHostToDevice);
    /*Crida al kernel*/
    CUDA_decode_mb_row <<<1, pc->mb_rows - 1 >>>(pbi, pc, xd);
    /*Recuperamos los datos de la GPU*/
    cudaMemcpy(gpu_pbi, pbi, sizeof (CUDA_VP8D_COMP), cudaMemcpyDeviceToHost);
    cudaMemcpy(gpu_pc, pc, sizeof (CUDA_VP8_COMMON), cudaMemcpyDeviceToHost);
    cudaMemcpy(gpu_xd, xd, sizeof (CUDA_MACROBLOCKD), cudaMemcpyDeviceToHost);
    /*Alliberem l'espai reservat a la GPU*/
    cudaFree(gpu_pbi);
    cudaFree(gpu_pc);
    cudaFree(gpu_xd);
}
```

Així doncs, el kernel `CUDA_decode_mb_row()` itera per a cadascun dels macroblocs existents a la filera i els descodifica. Cal destacar que la funció `CUDA_decode_macroblock_rows()` s'implementa com a una funció de C, ja que la implementació CUDA ha de ser transparent a la resta de la biblioteca. En canvi, el kernel `CUDA_decode_mb_row()` es declara com a `__global__` com es pot veure a l'algorisme 5.11.

`Algorisme 6.6 Declaració del kernel CUDA_decode_mb_row()`

```
__global__ void CUDA_decode_mb_row(VP8D_COMP *pbi, VP8_COMMON *pc, MACROBLOCKD *xd)
    [...]
}
```

A partir d'aquí totes les funcions necessàries són implementades com a kernels executables només per la GPU, amb l'identificador `__device__` i al nom original de la funció s'hi afegeix el prefix `CUDA_`. Tal i com s'ha explicat anteriorment, tots els tipus de dades i macros també s'han hagut de declarar com a codi de la GPU. En total són 1100, aproximadament, que s'han hagut de reescriure en CUDA C.

6.1.4.4 Modificacions fetes dins les funcions originals

Tal i com s'ha comentat abans, les funcions originals que són utilitzades per el kernel `CUDA_decode_mb_row(` s'han mantingut intactes, excepte en un únic aspecte. Ja s'ha explicat a l'apartat 5.2.3 el canvi en les declaracions de les interfícies afegint-hi la paraula clau `__device__` i el prefix `CUDA_` al nom de la funció, però dintre de les funcions necessàries per a la descodificació del fotograma

s'utilitzen funcions de C per a la gestió de memòria, són les funcions `memset()` i `memcpy()`. Tal i com també s'ha explicat al capítol 5.2.5, no es pot fer cap gestió de la memòria de la GPU a aquest nivell dintre d'un kernel, significa això que el codi no es pot reescriure en CUDA C? Si bé aquestes funcions no es poden usar en primera instància, es pot reescriure el codi sense haver-hi de cridar-les.

En el cas de la funció `memset()`, la seva sintaxi és la que es mostra a l'algorisme 6.7. Aquesta funció s'usa per inicialitzar N bytes de memòria, començant per l'apuntada per `ptr`, al valor 0.

Algorisme 6.7 Sintaxi de la funció `memset()`

```
memset(ptr, 0, N);
```

En canvi, la funció `memcpy()`, s'usa per duplicar zones de memòria. Tal i com es pot veure a l'algorisme 6.8, es copien N bytes de memòria començant per l'apuntat per `ptr_orig` a la zona de memòria apuntada per `ptr`.

Algorisme 6.8 Sintaxi de la funció `memcpy()`

```
memcpy(ptr, ptr_orig, N);
```

Per tant, si l'únic que fan aquestes funcions és copiar valors entre punters, això es pot implementar amb bucles que iterin N vegades sobre els punters a memòria corresponents. Si bé és una implementació menys eficient que la que usa les funcions `memset()` i `memcpy()` permet que el codi pugui ser compilat en CUDA C. Doncs, les crides a aquestes funcions es poden substituir per una implementació manual d'aquestes que facin la mateixa tasca, tal i com es mostra als algorismes 6.9 i 6.10.

Algorisme 6.9 Implementació manual de `memset()`

```
/*Implementació manual de memset(ptr, 0, N)*/
int i;
for(i = 0; i < N; i++){
    *(ptr + i) = 0;
}
```

6.1.4.5 Primeres proves i depuració

Una vegada fetes les modificacions al codi de la biblioteca libvpx es compila la biblioteca i el programa principal, *simple_ decoder* es compila amb la resta del codi font de la biblioteca libvpx i finalment s'enllaça amb la llibreria CUDA generada. Cal recordar que això s'ha fet per modificar el mínim de codi possible de la biblioteca libvpx original, d'aquesta manera, el codi escrit en C s'enllaça amb la biblioteca estàtica CUDA generada que es troba en llenguatge binari i no dóna cap problema de compatibilitat amb la resta del codi font.

Una vegada compilat tot el codi i generat el programa executable, es comencen les proves d'aquest. Tal i com s'ha explicat a la descripció del projecte (veure la subsecció 3.2.2) el codi de *simple_ decoder* requereix d'un arxiu IVF, doncs primer s'haurà de crear aquest fitxer. S'ha escollit un fitxer de vídeo pla en format Y4M, i per convertir-lo al format IVF s'ha usat el programa *vpx_ env* del SDK de libvpx usant la crida que es pot veure a l'algorisme 6.11.

La descripció dels paràmetres usats és la següent:

- `--ivf` : Indica al programa que el format de l'arxiu generat és IVF.

Algorisme 6.10 Implementació manual de `memcpy()`

```
/*Implementació manual de memcpy(ptr, ptr_orig, N)*/
int i;
for(i = 0; i < N; i++){
    *(ptr + i) = *(ptr_orig + i);
}
```

Algorisme 6.11 Conversió d'un vídeo al format IVF mitjançant `vpx_enc`

```
$>./vpxenc --ivf -w 352 -h 288 -o bus_cif.ivf bus_cif.y4m
```

- `-w 352 -h 288` : Indiquen la mida dels fotogrames del vídeo, la seva amplada i alçada respectivament.
- `-o bus_cif.ivf` : Arxiu de sortida.
- `bus_cif.y4m` : Arxiu d'entrada.

En aquest moment l'arxiu *bus_ cif.ivf* ja es troba llest per a que *simple_ encoder* el pugui agafar com a entrada del programa. Així doncs, es procedeix a executar la versió CUDA que s'ha desenvolupat que generarà un arxiu amb el mateix nom que el de l'entrada però amb extensió *yuv* amb la comanda:

```
$>./simple_decoder_lite bus_cif.ivf bus_cif_decoded.yuv
```

En aquest moment, el fitxer bus_cif_decoded.yuv conté el vídeo original descomprimit. Per tal de comprovar el resultat del vídeo descomprimit, es reprodueix el vídeo amb MPlayer amb la comanda per consola:

```
$>mplayer -demuxer rawvideo -rawvideo w=352:h=288:format=i420 bus_cif_decoded.yuv
```

El paràmetres que s'han usat per la crida a MPlayer es detallen a continuació:

- `-demuxer rawvideo` : Indica al reproductor que el vídeo a reproduir no està comprimit.
- `-rawvideo w=352:h=288:format=i420` : Són els paràmetres necessaris per reproduir el fitxer YUV, encapçalats per el paràmetre `-rawvideo`. Els paràmetres són la amplada i alçada del fotograma, i el format del medi.

Però, en el primer intent, el vídeo resultant és molt diferent al resultat esperat. Si bé s'ha accelerat la descodificació del vídeo, com es pot veure a la figura 6.5, el vídeo generat pel *simple_ decoder_ lite* és molt diferent a l'original.

A partir d'aquí, el desenvolupament de l'aplicació entra en una fase de depuració. Durant diversos dies s'ha estat analitzant el codi, depurant i consultant a la comunitat de desenvolupador del projecte WebM per a resoldre el problema. Però donat que el software per analitzar la viabilitat de la tecnologia CUDA és un objectiu secundari en aquest projecte, s'ha decidit prioritzar i desenvolupar el segon exemple d'aplicació de CUDA d'aquest projecte. Doncs, s'ha deixat la depuració de la biblioteca libvpx per a un treball futur ja que necessita molt més temps per entendre el codi font aliè a l'autor, i s'ha optat per desenvolupar el xifrador AES.

Figura 6.5: Fotogrames del vídeo original (esq.) i descomprimit per simple_decoder_lite (dreta)

6.2 El xifrador AES

El segon exemple que s'implementa en aquest projecte per a analitzar la viabilitat de la plataforma de NVIDIA és el xifrador AES. Ja s'ha fet una introducció a que és i en quins àmbits s'utilitza a la descripció del projecte (veure la secció 2.4). Abans d'entrar a explicar el funcionament de l'algorisme del xifrador AES, cal explicar alguns conceptes referents a la criptografia que ajudaran a entendre el funcionament d'AES encara que no es té com a objectius profunditzar-hi.

6.2.1 Tipus de xifradors

Actualment en criptografia existeixen diferents tipus de xifradors, però segons la manera com tracten les dades, s'agrupen dos grans grups:

Xifradors de flux El xifrador tracta les dades com a un flux, és a dir, les dades es tracten a nivell de byte o, fins i tot, a nivell de bit. S'utilitza un generador de flux clau que mitjançant una operació que varia a través del temps es barreja amb el flux de dades original per a produir el flux de dades xifrat. Aquest mètode és útil en casos que calgui xifrar informació que es crea i es consumeix en temps real com, per exemple, una conversa telefònica o una videoconferència.

Xifradors per blocs Els xifrador tracta les dades dividint-les en blocs d'una mida predeterminada abans de tractar-les. Aquesta tècnica s'usa amb xifradors de clau simètrica i l'algorisme que s'aplica és invariant.

En el mètode de xifrat per blocs s'ha explicat que s'utilitza en xifradors de clau simètrica. Un xifrador de clau simètrica no és res més que un xifrador on s'utilitza la mateixa clau per a xifrar les dades i per desxifrar-les.

6.2.2 Tipus de xifrat per blocs

Dintre del mètode de xifrat per blocs existeixen diferents variants a l'hora de tractar les dades, algunes més complexes que d'altres i que ofereixen diferents nivells de confidencialitat. En aquest a apartat s'exposen tots els tipus de tractament que pot fer AES sobre el bloc de dades. Així doncs, els diferents tipus de tractament per blocs són:

- Electronic Codebook (ECB)
- Cipher-Block Chaining (CBC)

- Propagating Cipher-Block Chaining (PCBC)
- Cipher Feedback (CFB)
- Output Feedback (OFB)
- Counter (CTR)

Electronic Codebook (ECB) En aquest mètode, els blocs dades són tractats de forma individual i independent amb una mateixa clau. Aquest és un mètode molt senzill però, en contrapartida, no és molt segur ja que per a un missatge en clar amb blocs de dades idèntics, aquests també seran idèntics en el text xifrat resultant. La figura 6.7 exposa el funcionament d'aquest mètode.

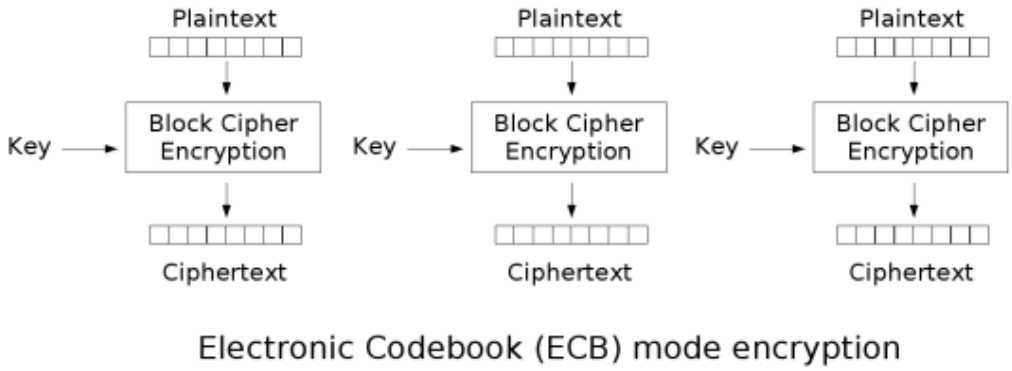

Figura 6.6: Tractament dels blocs segons el mètode ECB[7]

Cipher-Block Chaining (CBC) En aquest mode, al bloc de dades en clar se li aplica una operació XOR (representada amb el símbol $\oplus$) amb el bloc de dades xifrat anterior abans d'aplicar-li l'algorisme de xifrat AES. Això crea una dependència entre blocs, ja que fins que no s'hagi xifrat un bloc no es podrà xifrar el següent. En aquest sentit, es pot dir que transforma el xifrat per blocs en un xifrat de flux i, a més a més requereix d'un vector d'inicialització per a generar el primer bloc. A la figura 6.7 es pot veure gràficament com funciona aquest mètode.

Propagating Cipher-Block Chaining (PCBC) És una modificació del mode CBC. En aquest mètode el bloc de dades ja xifrat es combina amb el seu text pla corresponent i el resultat es combina amb el text pla del següent bloc abans de ser xifrat. Aquest mètode també transforma el xifrat per blocs en un xifrat de flux. També requereix d'un vector d'inicialització per generar el primer bloc.

Cipher Feedback (CFB) En aquest mètode, el bloc anterior ja xifrat es combina amb el bloc de dades en clar que s'està processant. El resultat és el text xifrat corresponent, que a més a més s'usa per a generar el següent bloc de dades xifrades. Aquest mètode també transforma el xifrat en un xifrat de flux, i també necessita d'un vector d'inicialització per a generar el primer bloc, com es pot veure a la figura 6.9.

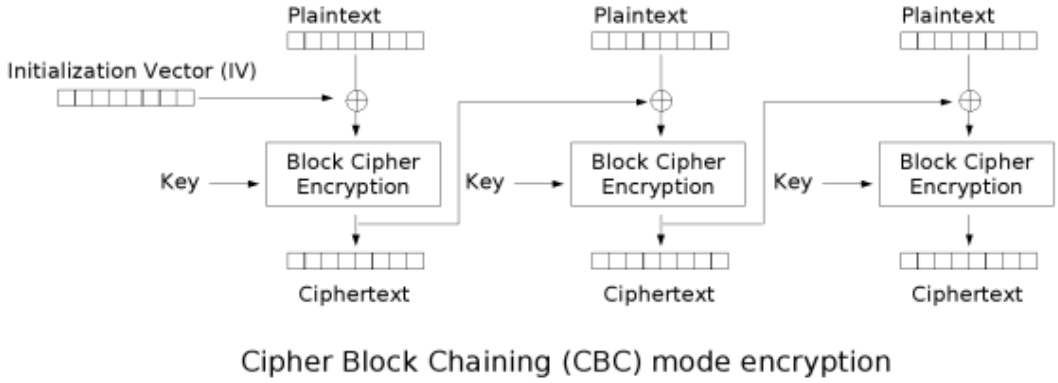

Figura 6.7: Tractament dels blocs segons el mètode CBC[7]

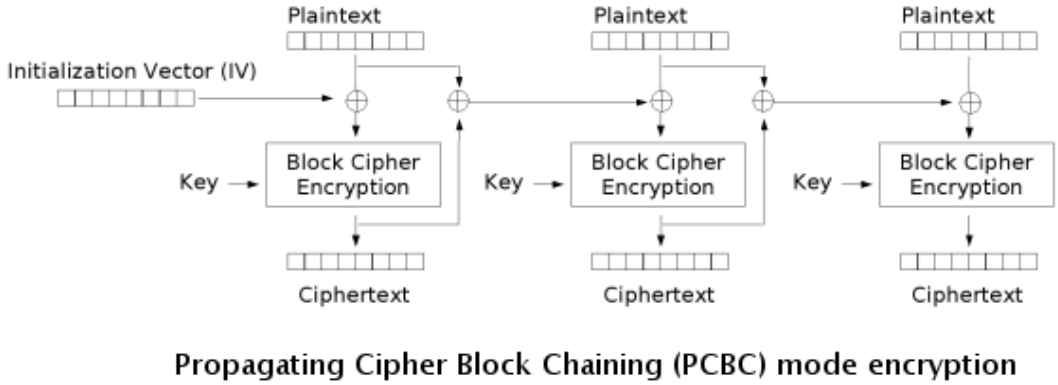

Figura 6.8: Tractament dels blocs segons el mètode PCBC[7]

Output Feedback (OFB) Aquest mètode és molt semblant a l'anterior, el CBC, però en aquest cas el bloc que serveix per a generar el següent bloc no és el bloc ja xifrat, si no que és la clau la que contínuament és xifrada per a la generació de cada bloc. Aquest mètode també transforma el xifrat en un xifrat de flux, i també necessita d'un vector d'inicialització per a generar el primer bloc. El seu funcionament es troba representat a la figura 6.10.

Counter (CTR) Aquest mètode utilitza un comptador que es xifra amb la clau i després es combina amb el bloc de text clar per a generar el bloc xifrat. Cal que el comptador sigui incremental i que no repeteixi els seus valors durant un llarg període de temps, normalment un comptador incremental simple és l'opció més comuna a utilitzar en aquest mètode. La figura 6.11 il·lustra el funcionament segons aquest mètode.

El que es pretén en aquest projecte és paral·lelitzar l'algorisme AES, per tant els mètodes CBC, PCBC, CFB i OFB no es poden fer servir ja que trenquen el paral·lelisme que es vol assolir al transformar el xifrat per blocs en un xifrat de flux. Així doncs, les úniques opcions que resten són el mètode ECB i el mètode CTR.

AES és un estàndard, i per tal que l'aplicació resultant sigui totalment compatible amb qualsevol altre software que implementi el mateix algorisme s'ha desenvolupat el xifrador seguint els exemples que apareixen als apèndix de la seva especificació. En aquesta especificació de

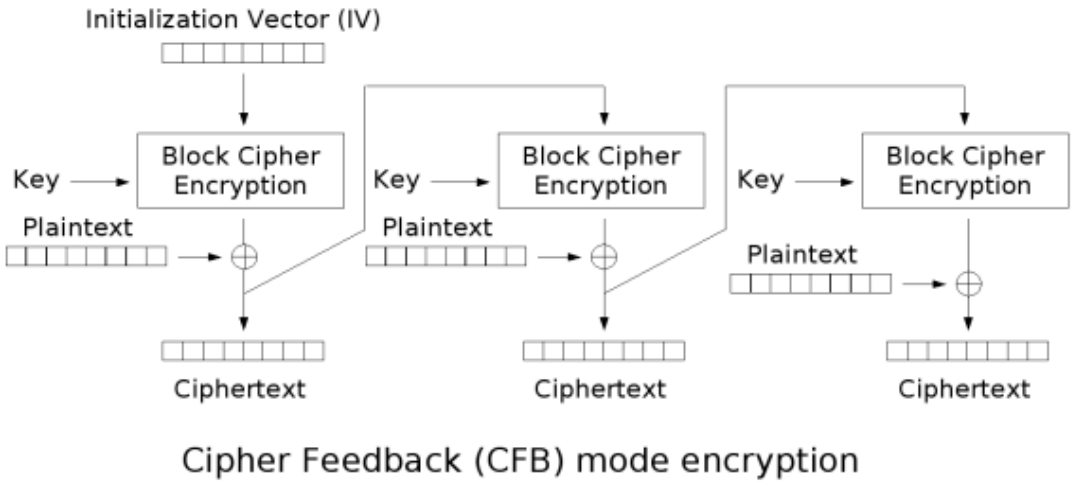

Figura 6.9: Tractament dels blocs segons el mètode CFB[7]

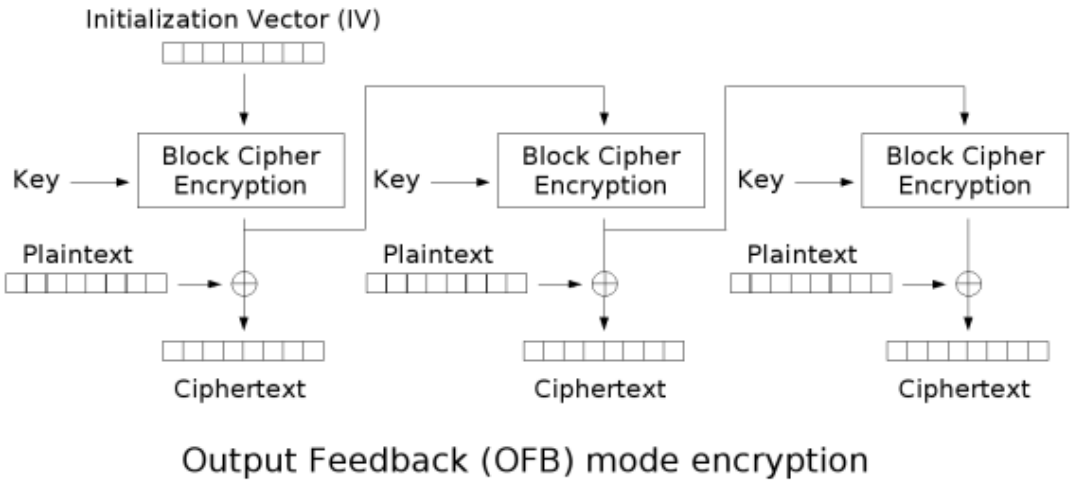

Figura 6.10: Tractament dels blocs segons el mètode OFB[7]

l'estàndard AES només es tracta el mètode ECB, que és el mètode usat a l'especificació de l'estàndard AES. Per això es deixa la implementació del mètode CTR per a treball futur, ja que corregeix la debilitat del mètode ECB i és igualment paral·lelitzable amb CUDA.

6.2.3 L'algorisme del xifrador

Ara que ja està decidit el tipus de tractament que es farà dels blocs cal explicar com funciona el xifrador AES, encara que no s'explicarà en detall en aquest projecte.

Aquest xifrador té una estructura bastant simple, consta de rondes de quatre etapes per les quals s'itera un nombre determinat de vegades especificat a l'estàndard, a més d'aplicar unes certes etapes abans. El nombre de rondes depèn de la mida de la clau donada al xifrador, que donen lloc als tres casos següents:

- Clau de 128 bits (16 Bytes) correspon a 10 rondes.
- Clau de 192 bits (24 Bytes) correspon a 12 rondes.
- Clau de 256 bits (32 Bytes) correspon a 14 rondes.

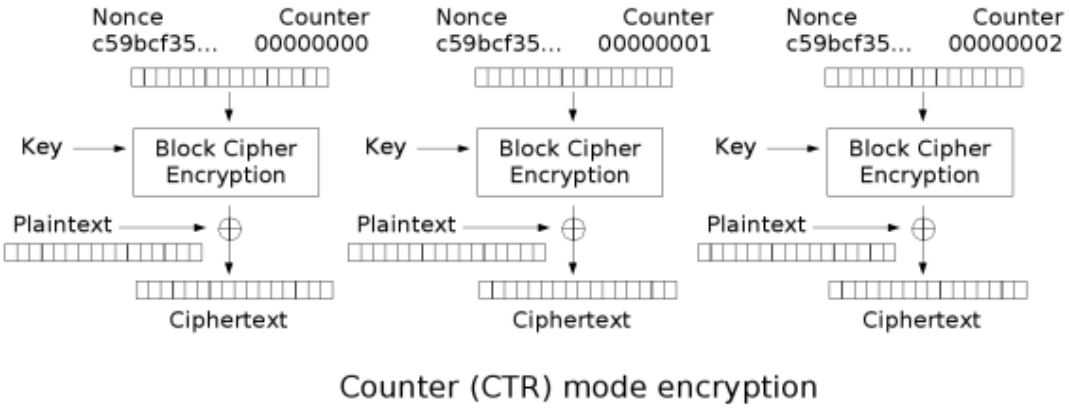

Figura 6.11: Tractament dels blocs segons el mètode CTR[7]

Llavors l'esquema sencer queda tal i com es mostra a la figura 6.12, on es pot veure que en el procés de xifrat per a una clau de 128 bits, abans de la primera ronda s'aplica a les dades l'etapa anomenada `AddRoundKey`. Es pot veure que el procés de desxifrat de les dades consisteix en aplicar-hi les inverses de cadascuna de les etapes en l'ordre invers al xifrat.

Com ja s'ha explicat, AES és un algorisme de xifrat per blocs, doncs, l'algorisme que es veu a la figura 6.12 s'aplica a un bloc de dades. Aquest bloc de dades són 16 Bytes del missatge original, ja sigui xifrat o en clar, i a aquest bloc de dades se l'anomena matriu *State*.

En la implementació realitzada per a aquest projecte, el xifrador segueix el pseudocodi que es mostra a l'algorisme 6.12. El programa conté un buffer que pot emmagatzemar un volum de dades de aproximadament 512MB que carrega i tracta (xifrant o desxifrant segons indiqui `Mode`) mentre hi hagin dades per llegir del fitxer d'entrada indicat i sobreescriu en el mateix buffer les dades ja tractades. La decisió de crear un buffer de gairebé 512MB té la seva causa a la implementació de la versió CUDA del xifrador.

Algorisme 6.12 Pseudocodi de l'algorisme AES implementat

```
AES_Inicialitza()
Mentre (CarregarBuffer()) fer
    Per (cada matriu State al buffer) fer
       AES_Encripta(State, Mode)
    fPer
    EscriureBuffer()
fMentre
AES_Finalitza()
```

6.2.3.1 L'expansió de la clau

Com es pot veure a la figura 6.12, a l'etapa `AddRoundKey` s'utilitza un element anomenat *clau expandida*. Aquesta clau es genera a partir de la clau donada al xifrador que ocupa els primers bytes de la clau expandida. La resta dels bytes fins a arribar a la longitud corresponent s'omplen seguint un algorisme d'expansió, especificat a l'estàndard AES.

Un detall important es la longitud d'aquesta clau, cada vegada que s'executa l'etapa `AddRoundKey` usa una paraula de quatre bytes de la clau expandida. Doncs, la longitud d'aquesta clau expandida depèn del nombre de rondes, per tant, de la longitud de la clau donada al xifrador. S'arriba a tres longituds diferents per a aquesta clau expandida:

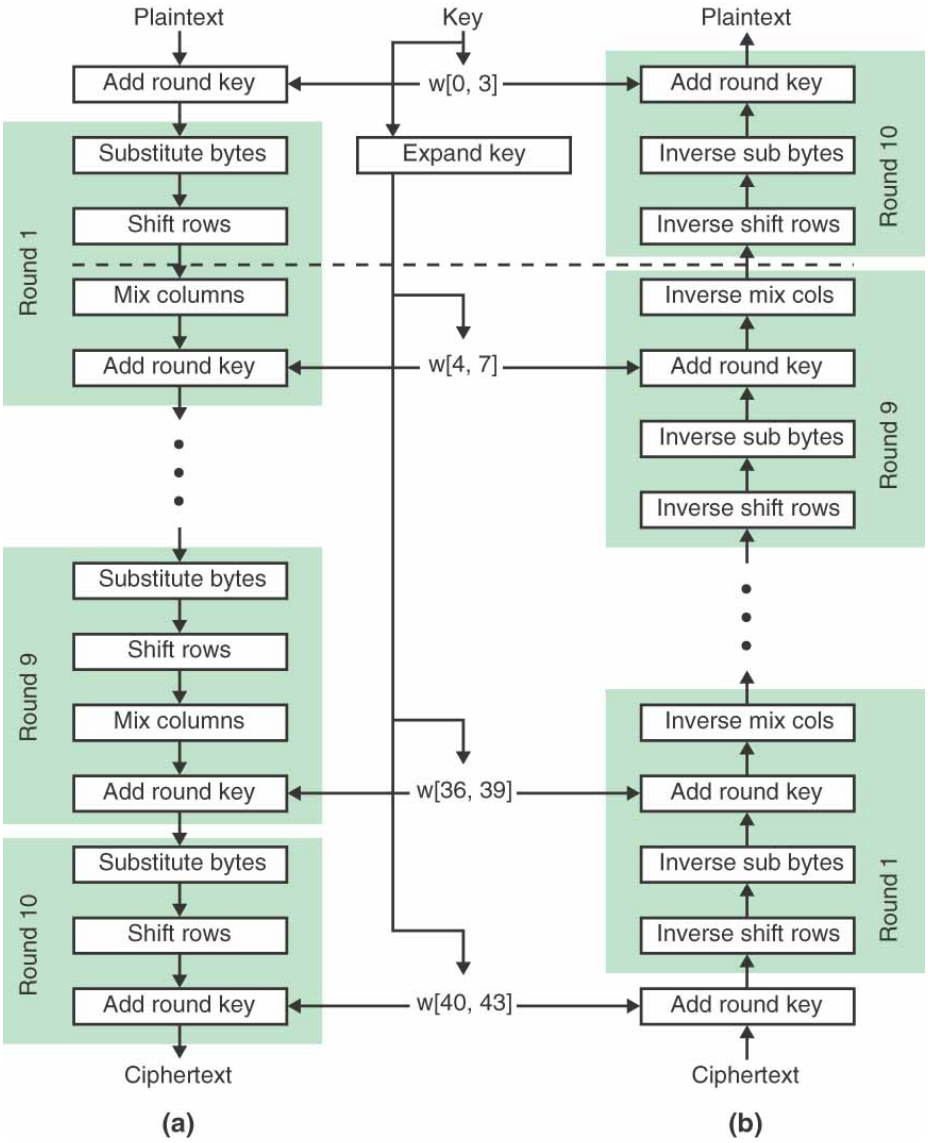

Figura 6.12: Esquema general de l'algorisme AES[25]

- Per a claus de 128 bits, la clau expandida té una longitud de 44 paraules de 4 bytes.
- Per a claus de 192 bits, la clau expandida té una longitud de 52 paraules de 4 bytes.
- Per a claus de 256 bits, la clau expandida té una longitud de 60 paraules de 4 bytes.

Una vegada la clau original ha estat expandida, el xifrador aplica les rondes necessàries al bloc que està tractant, i en cada ronda aplica una sèrie d'etapes. Aquestes etapes són les quatre següents:

- `AddRoundKey`
- `Substitute bytes`
- `Shift Rows`
- `Mix Columns`

6.2.3.2 L'etapa `AddRoundKey`

En aquesta etapa, a la matriu *State* se li aplica una operació XOR amb les quatre paraules corresponents (16 Bytes), segons la ronda en la que es trobi el xifrador, de la clau expandida. A la figura 6.13 es pot veure gràficament aquest procés.

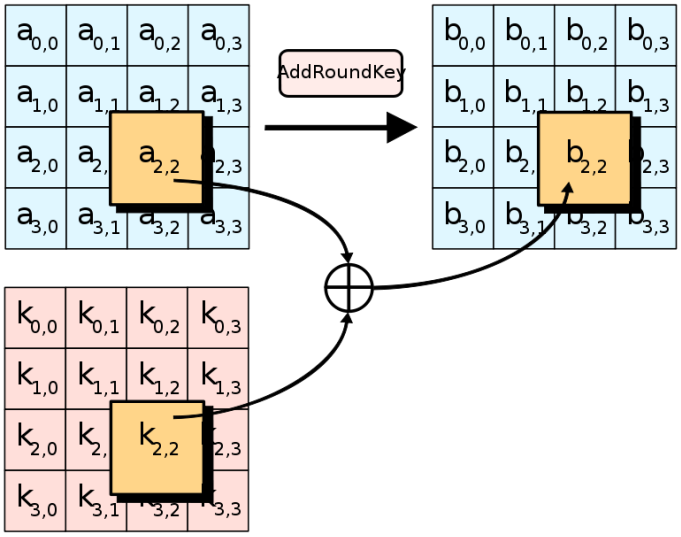

Figura 6.13: Etapa `AddRoundKey`[6]

En el desxifrat, la inversa d'aquesta etapa no existeix, ja que la inversa de l'operació XOR, és aplicar-hi un altre cop la mateixa operació XOR.

6.2.3.3 L'etapa `Substitute bytes`

En aquesta etapa, cada byte de la matriu *State* és substituït per el seu byte corresponent dins una taula predefinida anomenada *S-Box*. Aquesta taula està dissenyada expressament per a donar més seguretat al xifrador, i la forma d'aplicar-la és senzilla. De cada byte de la matriu *State*, els quatre bytes de més pes s'interpreten com la coordenada X a consultar de la *S-Box* i els quatre de menys pes s'interpreten com la coordenada Y. El valor original de cada byte de la matriu es substitueix pel valor trobat a la *S-Box*, la figura 6.13 il·lustra aquesta etapa, on la *S-Box* està representada per la caixa amb el símbol *S*. El contingut tant de la *S-Box* com el de la *S-Box* inversa no s'exposen ja que no aporten cap valor afegit.

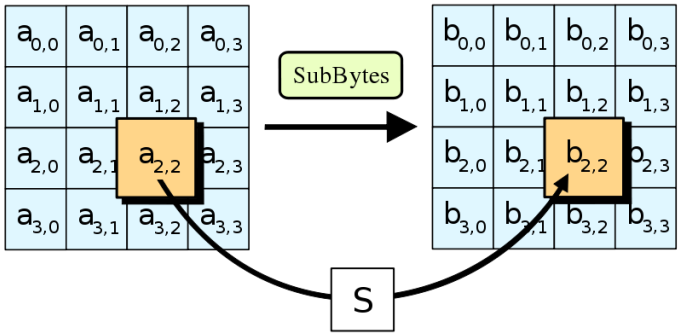

Figura 6.14: Etapa `Substitute bytes`[6]

En el procés de desxifrat s'aplica una *S-Box* inversa que amb el mateix procés d'interpretar els bytes com a coordenades desfà el procés efectuat en aquesta etapa durant el xifrat.

6.2.3.4 L'etapa `Shift Rows`

Aquesta etapa és molt simple, a la matriu *State* es roten cadascuna de les files tantes posicions a l'esquerra com el número de fila sigui, és a dir, la fila 0 no es modifica, la fila 1 es desplaça una posició, i així successivament. La figura 6.15 representa gràficament aquesta etapa.

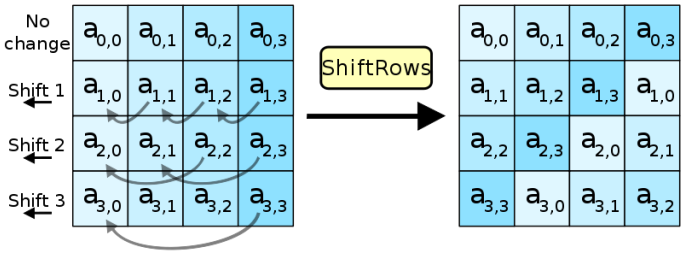

Figura 6.15: Etapa `Shift Rows`[6]

En el desxifrat, la inversa d'aquesta etapa consisteix en desplaçar les files en sentit contrari a com es fa al xifrat.

6.2.3.5 L'etapa `Mix Columns`

Aquesta és l'etapa més complexa de tot el xifrador, encara que en aquest projecte s'ha optat per una sol·lució més directa, però primer cal explicar en que consta aquesta etapa.

En l'etapa `Mix Columns`, es multipliquen dues matrius. Aquestes dues matrius són cadascuna de les columnes de la matriu *State* multiplicada per la matriu corresponent (segons si s'està xifrant o desxifrant) de la figura 6.16.

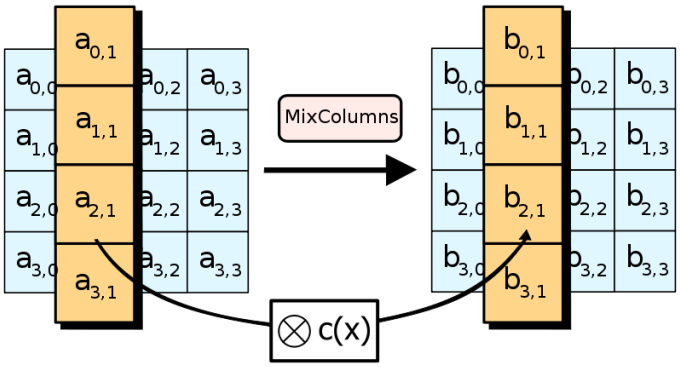

Figura 6.16: Etapa `Mix Columns`[6]

Les matrius corresponents al xifrat i desxifrat que es multipliquen per cada una de les columnes són les que apareixen a la figura 6.17.

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \quad \begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix}$$

Figura 6.17: Matrius per al xifrat (esquerra) i desxifrat (dreta) a l'etapa `Mix Columns`[6]

El problema arriba a l'hora de fer aquestes operacions. La multiplicació de matrius no és difícil d'implementar en software, però el problema apareix perquè aquestes operacions s'han de fer en un camp finit $GF(2^8)$. Per resoldre aquest problema, la multiplicació de les matrius s'implementen amb l'ajuda d'unes taules predefinides per a treballar en aquest camp finit, tant

al xifrat com al desxifrat. Amb aquestes taules, la implementació de la multiplicació es pot realitzar com una multiplicació de matrius bàsica, tal i com mostra l'algorisme 6.13 encara que les sumes de l'algorisme de multiplicació de matrius original es substitueix per operacions XOR. En aquest algorisme es multiplica una columna de la matriu *State* per la matriu corresponent al procés de xifrat i es guarda en un vector temporal `tmp[0]` que es tracta més endavant.

Algorisme 6.13 Implementació de la multiplicació tabulada de matrius en el camp finit $GF(2^8)$

```
tmp[0] = GM2[State[fila][0]] ^ GM3[State[fila][1]] ^
         State[fila][2] ^ State[fila][3];
tmp[1] = State[fila][0] ^ GM2[State[fila][1]] ^
         GM3[State[fila][2]] ^ State[fila][3];
tmp[2] = State[fila][0] ^ State[fila][1] ^
         GM2[State[fila][2]] ^ GM3[State[fila][3]];
tmp[3] = GM3[State[fila][0]] ^ State[fila][1] ^
         State[fila][2] ^ GM2[State[fila][3]];
```

6.2.4 La versió CUDA del xifrador

6.2.4.1 Declaracions de les funcions i objectes a la GPU

Partint de la versió del xifrador AES implementat per a CPU, s'ha desenvolupat la versió GPU adaptant el codi font a CUDA C. Tal i com s'ha fet amb la biblioteca libvpx a la secció 6.1.4.3, les funcions que implementen les quatre etapes de cada ronda s'han definit com a kernels `__device__` els següent elements:

- Les implementacions de les quatre etapes del xifrador AES.
- Les definicions de la *S-Box* i *S-Box* inversa.
- La funció auxiliar encarregada de fer la rotació de les files de la matriu *State* a l'etapa `Shift Rows`.
- Les taules necessàries per a les multiplicacions de matrius a l'etapa `Mix Columns`.

Per una altra banda, com la funció per a l'expansió de la clau no es pot paral·lelitzar es deixa tal qual s'implementa en la versió CPU del xifrador. Però en aquesta tasca també intervenen la funció per a la rotació de les files de la matriu *State* i la matriu *S-Box*. Per solucionar aquesta necessitat es duplica la declaració d'aquests elements canviant el seu nom per a diferenciar-les de les que ha d'usar la CPU. Per exemple la *S-Box* queda declarada dues vegades de la següent forma afegint-hi el prefix `CUDA` al nom:

- `const SBOX[256]` per a ser usada per la CPU.
- `__device__ const CUDASBOX[256]` per a ser usada per la GPU.

A més a més, el kernel que és cridat per la CPU és la funció `AES_Encripta`, per tant aquesta funció s'ha de redefinir com a un kernel declarat com a `__global__` per a que pugui ser cridat per la CPU. Doncs, les diferències en la definició de les funcions són les que es mostren a l'algorisme 6.14.

L'única funció que en la que s'ha de modificar codi, és el kernel `AES_Encripta()`, on s'ha de generar l'identificador del fil, crear el control de flux per a fils desocupats i implementar la sincronització dels fils amb la funció `__syncthreads()` tal i com es mostra al pseudocodi 6.15.

Algorisme 6.14 Exemple de les declaracions

```
/*Interficies de les funcions originals de la versió CPU*/
void AddRoundKey(uChar State[4][4], uChar ronda)
void AES_Encripta(uChar State[4][4], uChar inv)
const uChar GM2[256]
/*Interficies de les funcions de la versió GPU*/
__device__ void AddRoundKey(uChar State[4][4], uChar ronda)
__global__ void AES_Encripta(uChar State[4][4], uChar inv)
__device__ const uChar GM2[256]
```

A més a més, al kernel `AES_Encripta()` rep per paràmetre: el nombre de matrius `States` que s'han llegit del disc i carregat al buffer, els punter a la clau expandida i buffer de dades que s'han copiat a la memòria de la GPU, el nombre de rondes que cal aplicar en l'algorisme del xifrador i el mode de funcionament (xifrat o desxifrat).

Algorisme 6.15 Pseudocodi de la implementació del kernel `AES_Encripta()`

```
AES_Encripta(..., NombreStatesAlBuffer, Mode)
    fil_idx <- blockIdx.x * blockDim.x + threadIdx.x
    Si (fil_idx < NombreStatesAlBuffer) fer
        CopiaStateDelBuffer()
        TractaState(Mode)
        CopiaStateAlBuffer()
    fSi
    __syncthreads()
fiAES_Encripta
```

6.2.4.2 La configuració de la crida al kernel

Una vegada que ja s'han tornat a declarar les funcions com a kernels CUDA i els vectors necessaris a la GPU, cal implementar la crida al kernel, tota la gestió de la memòria de la GPU, i la transferència de les dades.

La primera d'aquestes tasques s'implementa amb les funcions `cudaMalloc()` i `cudaMemcpy()` que ja s'han vist a l'apartat 5.2.4, i els elements que cal enviar a la GPU són la clau expandida i el buffer de dades. Després d'això es fa la crida al kernel `AES_Encripta()`. Però, quina ha de ser la configuració de la crida al kernel?

Si es vol que aquesta versió CUDA del xifrador sigui compatible també amb les targes gràfiques amb una capacitat de comput inferior a la 1.2, el nombre de fils d'execució ha de ser de 512 com a màxim. I pel que fa al nombre de blocs, totes les targes gràfiques existents tenen un valor màxim de 65535. Ara, caldrà calcular quina quantitat de dades permet tractar una configuració de 65535 blocs i 512 fils d'execució per bloc, això es pot calcular amb la fórmula següent:

$$65535\,Blocs \cdot \frac{512\,Fils\,d'execució}{1\,Bloc} = 33553920\,Fils\,d'execució$$

Llavors, amb aquesta configuració es creen un total de 33553920 fils d'execució a la GPU. Si es té en compte que cada fil d'execució haurà de tractar una matriu *State*, tenim doncs que es poden tractar 33553920 matrius *State* amb una sola crida al kernel `AES_Encripta()`. Però

cal tenir present quina quantitat de dades representa aquesta xifra, ja que potser sobrepassi la quantitat de memòria disponible a la GPU. Per tant, es calcula el volum de dades que correspon als 33553920 fils amb la fòrmula següent:

$$33553920\,Fils\,d'execució \cdot \frac{16\,Bytes}{1\,Fil\,d'execució} \cdot \frac{1KB}{1024Bytes} \cdot \frac{1MB}{1024KB} = 511,9921875MB$$

Doncs, amb la crida esmentada anteriorment de 65535 blocs i 512 fils d'execució per bloc permet tractar gairebé 512MB de dades, una quantitat considerablement gran. Encara que per al hardware que s'usa en aquest projecte hi ha espai suficient, caldrà veure si en un dispositiu amb només 512MB de memòria és capaç d'emmagatzemar aquesta quantitat realment, o bé passarà com en el cas del dispositiu usat en aquest projecte on la memòria es d'1GB però el valor real de memòria disponible està per sota d'aquest valor.

Cal tenir en compte també que a diferència de la CPU, la GPU té diferents memòries. En el cas de la implementació d'aquest xifrador, tant les dades que s'envien a la GPU com les dues S-BOX i les taules per a la multiplicació de matrius s'emmagatzemen a la memòria global i això resta espai a les dades que es poden tractar. Així que es pot aventurar a dir que segurament als dispositius compatibles amb CUDA amb una memòria total de 512MB s'haurà d'ajustar a la baixa el nombre de fils d'execució que es poden llançar a la GPU.

Ara que ja està implementada la crida al kernel, cal implementar la gestió de la memòria i l'intercanvi de dades. Primer de tot, cal definir quina serà la mida del buffer que usarem a la memòria principal del sistema per a guardar les dades llegides del disc dur, enviar-les a la GPU, rebre les dades tractades i escriure-les al disc dur.

Si ja s'ha deduït que la configuració de 65535 blocs i 512 fils d'execució per bloc pot tractar 33553920 matrius *State*, doncs la mida d'aquest buffer serà de:

$$33553920\,Fils \cdot \frac{16\,Bytes}{1\,Fil} = 536862720\,Bytes$$

Encara que la configuració final de la crida sigui de 33553920 fils d'execució, per tal de depurar el codi i veure que funciona correctament, es prova d'executar només amb 1 bloc i 1 fil d'execució. D'aquesta manera s'emula el funcionament de la versió CPU a la GPU, i una vegada verificat el correcte funcionament de l'algorisme d'aquesta configuració, es torna a compilar el programa amb els valors finals de 65535 bloc i 512 fils per bloc.

Només queda implementar les funcions de reserva d'espai, transferència de dades i alliberament de memòria i la versió CUDA del xifrador AES ja estarà acabada. Llavors el pseudocodi de l'algorisme CUDA d'AES queda tal i com es mostra a l'algorisme 6.12. Les equivalències amb la implementació en CUDA C d'aquest pseudocodi són:

`ReservaEspaiGPU()` s'implementa amb la funció `cudaMalloc()`. Caldrà fer la reserva per a dos elements, la clau expandida i el buffer de dades.

`EnviarDadesAGPU()` s'implementa amb la funció `cudaMemcpy()` amb la constant `cudaMemcpyHostToDevic` Cal enviar la clau expandida i el buffer de dades llegit a la iteració actual, encara que la clau expandida només cal enviar-la una vegada ja que no variarà.

`RecollirDadesDeGPU()` s'implementa amb una altra crida a la funció `cudaMemcpy()` amb la constant `cudaMemcpyDeviceToHost`. En aquest cas només cal copiar les dades tractades, no ens cal la clau expandida, ja que no s'usa fora del kernel.

`AlliberarEspaiGPU()` s'implementa amb la funció `cudaFree()`. Cal alliberar les dues reserves d'espai fetes anteriorment, la de la clau expandida i la del buffer de dades.

Algorisme 6.16 Pseudocodi de la funció principal de la versió CUDA d'AES

```
Main(){
    AES_Inicialitza
    ReservaEspaiGPU()
    Mentre(CarregarBuffer()) fer
        EnviarDadesAGPU()
        AES_Encripta<<<65535, 512>>>()
        RecollirDadesDeGPU()
    fMentre
    AlliberarEspaiGPU()
fMain
```

6.2.4.3 El suport d'arxius de gran volum

Una vegada el xifrador ja està implementat cal veure si aquest pot tractar arxius voluminosos. Per arxius voluminosos es refereix a arxius que ocupin diversos gigabytes d'espai en disc.

Les primeres proves realitzades amb un arxiu de 4.7 GB han resultat en que el programa no escriu més de 2 GB en l'arxiu de sortida. Això es degut a les funcions usades per al tractament dels arxius.

En el xifrador s'usen funcions estàndards de C per a llegir l'arxiu i per a crear l'arxiu de sortida. La funció que s'encarrega d'obrir l'arxiu es `fopen()` i aquesta funció no suporta arxius d'un volum superior a 2 GB. No s'entra a explicar com s'implementa aquesta funció ja que per a aquest projecte no aporta cap valor afegit, llavors com es poden tractar el arxius de gran volum?

Per a tractar aquesta mena d'arxius s'usa la funció depèn de quin tipus de sistema operatiu s'estigui usant, si és un entorn Windows o un GNU/Linux (com Ubuntu). Segons quin dels dos executi l'aplicació es farà el següent:

- Per a un entorn Windows no cal fer res més que definir la constant `_FILE_OFFSET_BITS 64` al començament de tot el codi font. El funcionament de la funció `fopen()` per a arxius grans és transparent per al desenvolupador.
- Per a un entorn Unix, a més a més de declarar la constant `_FILE_OFFSET_BITS 64` al començament del codi font, cal usar la funció `fopen64()` quan es vulgui obrir un arxiu en substitució de la funció `fopen()` original, i no cal modificar cap paràmetre a la seva crida ja que el seu funcionament és idèntic.

Com el desenvolupament s'ha fet en Ubuntu, un sistema operatiu Unix, s'ha modificat la crida a la funció `fopen()` per `fopen64()`. A més, de incloure `#define _FILE_OFFSET_BITS 64` al començament del codi font. Una vegada fet això, el xifrador ha pogut tractar l'arxiu de 4.7 GB sense cap problema.

6.2.4.4 La compilació a Windows 7

El codi del xifrador no usa cap llibreria de sistema, es pot dir que el codi implementat tant per la versió CPU com per a la versió GPU són independents del sistema operatiu on es compili. Tret, és clar, de la gestió de la compatibilitat amb els arxius de gran volum comentada en l'apartat anterior.

Quan s'ha afegit el codi del xifrador AES a un projecte de C++ dins l'IDE Visual Studio, aquest l'ha pogut compilar sense cap problema, tret d'algunes declaracions de variables locals que no estaven declarades al començament d'una funció i això feia que el compilador de Visual

Studio mostrés errors. Però una vegada mogudes aquestes declaracions al començament de les respectives funcions, el codi ha compilat sense problemes, encara que hi apareixen alguns avisos indicant que la funció `fopen()` és obsoleta i que s'usi `fopen_s()` que és més segura.

Aquests avisos no són cap problema greu, ja que `fopen()` es pot usar sense cap tipus de problema. El que les altres funcions serveixin només per a Windows implica modificar el codi per a adaptar-lo al sistema operatiu sobre el que es compili, i per solucionar aquest avís que dóna el compilador de Visual Studio només s'ha d'afegir la línia `#define _CRT_SECURE_NO_DEPRECATE` al començament del codi.

Com que la definició d'aquest avís i la gestió dels arxius grans varien segons es compili el codi font en Windows o Ubuntu, es defineixen unes etiquetes de preprocessador per a que el compilador, abans de compilar el codi font apliqui una modificació o una altra segons el sistema. Això es fa amb el condicional de preprocessador `#ifdef-#else-#endif`, per tant, el codi modificat queda com es mostra a l'algorisme 6.17. Com a la plataforma Windows l'etiqueta de preprocessador `_WIN32` es reservada i s'usa a la compilació dintre Visual Studio, es pot usar per identificar si el sistema on s'està compilant el codi font és un entorn Windows o GNU/Linux. D'aquesta manera si es compila en un Windows s'aplicarà la constant per a solucionar els avisos que provoca la funció `fopen()`, i si no es modifica el nom `fopen()` per a que quan en GNU/Linux es faci la crida a aquesta funció, realment cridi a `fopen64()`.

Algorisme 6.17 Codi de preprocessador per adaptar-lo segons el sistema operatiu

```
#ifdef _WIN32
#define _CRT_SECURE_NO_DEPRECATE
#else
#define fopen fopen64
#endif
```

Capítol 7

Proves del software i guany del projecte

7.1 Introducció

Tal i com s'ha explicat al capítol anterior la modificació de la biblioteca libvpx no s'ha arribat a completar satisfactòriament, per tant, les proves de rendiment es fan només sobre el xifrador AES a l'entorn que està descrit a la descripció d'aquest projecte (veure l'apartat 3.2.2).

Una vegada s'ha depurat el codi i revisat el seu funcionament, es procedeix a fer les proves de rendiment. En aquestes proves es mesura el temps que triga el programa en tractar les dades, el seu consum energètic i com afecta aquest a la temperatura dels dispositius.

Tal i com s'explica al capítol dedicat a la secció 2.4.1 del projecte, la versió CPU s'executa també en un model i7 920. Per tal que les proves en aquesta CPU sigui el més semblant possible a les proves fetes en la CPU model Core2Duo E6600, s'ha agafat el disc dur amb el sistema on s'han fet les proves en el model E6600 i connectat a la màquina amb el i7 920. D'aquesta manera les proves es fan sobre el mateix sistema operatiu.

Per a realitzar aquestes proves s'han usat dos programes per a mesurar les temperatures: Xsensors, una aplicació disponible per a Ubuntu, per a mesurar la temperatura de la CPU i el propi controlador de NVIDIA per a mesurar la temperatura de la GPU. A més a més, els arxius que s'han usat en aquestes proves (de 128, 256, 384 i 512MB) s'han generat amb un altre petit programa pròpi implementat en C++, que usant el fitxer de 16 bytes usat per a la depuració del xifrador genera aquests fitxers.

Finalment, per fer aquestes proves d'una manera més automatitzada, s'han fet un parell de scripts per a la consola de comanda d'Ubuntu. Aquests scripts (un per a cada versió del xifrador) executen el programa Xsensors i el controlador de NVIDIA per veure les temperatures i després iteren l'execució del xifrador per a cadascun dels arxius de proves i guarden en un fitxer tot el que aquests imprimeixen per pantalla, per així tenir un registre escrit del que mostren. El codi del script per a la versió CPU es mostra a l'algorisme 7.1, i el de la versió GPU només es modifica el nom de l'executable pel corresponent a la versió GPU del xifrador. La línia de comandes que s'usa per a executar el xifrador es troba explicada al manual d'usuari del software.

La nomenclatura utilitzada per a les taules que es veuen en els apartats següents segueixen la següent nomenclatura:

- C2D representa la CPU Intel Core 2 Duo E6600
- i7 representa la CPU Intel Core i7 920
- GPU representa el hardware CUDA, la GPU GTX550Ti

Algorisme 7.1 Script per a l'automatització de les proves

```
#!bin/sh
FILES=/home/ruben/Escritorio/generated_files/*
xsensors & nvidia-settings --page="Thermal Settings" &
echo "Ejecutando AES CPU"
for i in $FILES do
    echo "Procesando $i"
    ./aes e 3 "$i" "$i.outputAES" "Key" > $i.logAES
done
```

7.2 Temps d'execució

La primera comparativa que cal fer per veure si la tecnologia CUDA és viable és la referent al temps que triguen totes dues (la versió CPU i la versió CUDA) en processar les dades carregades al buffer. Per tant, no es mesura la resta de temps, com és el temps que triga el programa a llegir i escriure les dades al disc dur ja que aquest temps serà el mateix en ambdues versions del xifrador.

Per a realitzar aquesta mesura de temps, s'ha decidit implementar un comptador intern al propi algorisme del xifrador. S'emmagatzema en una variable el moment en que l'algorisme comença a tractar les dades usant la funció `clock()` estàndard de C que es troba dintre la biblioteca *time.h* i després, aquest valor es compara amb el moment en que s'ha completat el tractament de totes les dades del buffer. Aquesta diferencia de temps, en segons, indica els temps que triga el programa en tractar les dades. El pseudocodi 7.2, explica com s'inclou aquest mecanisme dins l'algorisme del xifrador i s'imprimeix per pantalla la diferencia de temps. Cal dir que en el cas de la versió GPU, la mesura del temps de càlcul inclou l'enviament, el tractament i la recepció de les dades tractades.

Algorisme 7.2 Pseudocodi de la mesura del temps de tractament del buffer

```
AES_Inicialitza()
Mentre (CarregarBuffer()) fer
    temps <- clock()
    Per (cada matriu State al buffer) fer
        AES_Encripta(State, Mode)
    fPer
    ImprimirPerPantalla(clock()-temps)
    EscriureBuffer()
fMentre
AES_Finalitza()
```

Una vegada el codi ja està adaptat per a que mostri el temps que triga en tractar el buffer, s'executen les proves amb els scripts. Una vegada executades les proves, les dades recollides van ser les mostrades a la taula 7.2, on les columnes són els processadors i les files les diferents mides dels arxius de proves usats, i a la gràfica de la figura 7.2 es pot veure més fàcilment la diferència entre els resultats.

Encara que en cada una de les quatre execució el temps de tractament varia, així que s'ha fet una mitjana amb tots els valors resultants. Encara que aquesta variació no arribava al mig segons en cap dels tres casos, s'han de considerar aquestes dades com a orientatives.

Es pot veure una dada curiosa entre els resultats de les dues CPU usades, el processador E6600 i el i7 920. Entre ells dos hi ha gairebé cinc anys de diferència, temps en el qual el

	C2D	i7	GPU
128 MB	77.15 s	82.56 s	1.32 s
256 MB	157.34 s	165.22 s	1.67 s
384 MB	240.06 s	247.89 s	1.99 s
512 MB	320.09 s	330.52 s	2.31 s

Taula 7.1: Temps de tractament dels diferents fitxers de proves

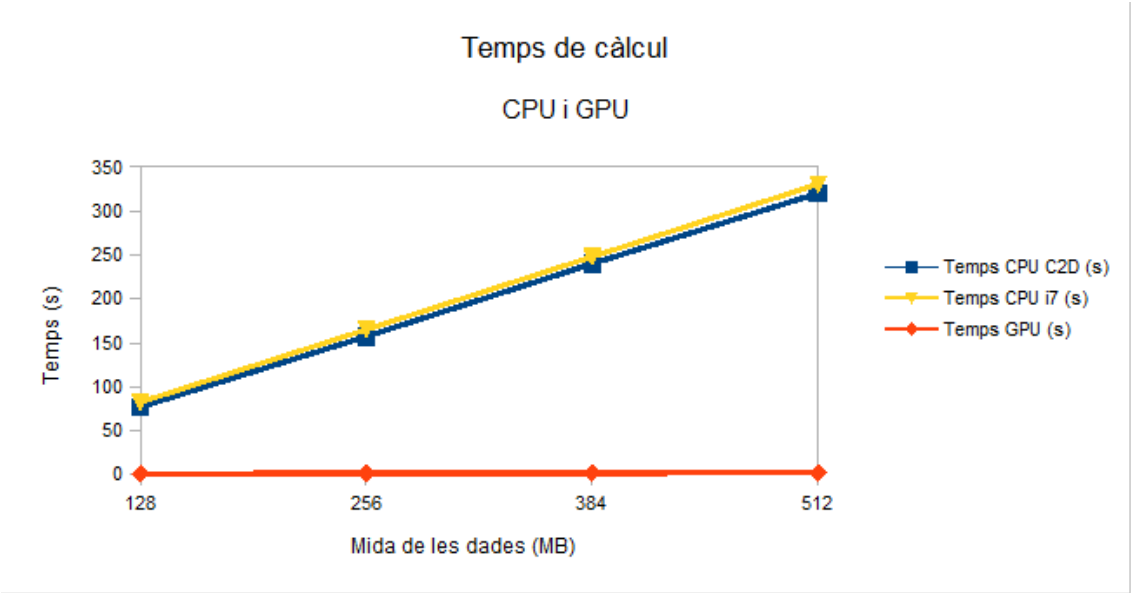

Figura 7.1: Representació gràfica dels temps de tractament de les dades

rendiments dels processadors ha millorat molt. Però cal tenir en compte que la versió CPU del xifrador AES desenvolupat només usa un únic nucli de la CPU i pel que fa a les dades recollides durant les proves, un sol nucli d'un processador de gairebé última generació és igual o, fins i tot, menys ràpid que un de fa cinc anys. Caldrà veure si en altres aspectes el Core i7 treu avantatge sobre el E6600.

Pel que fa al temps d'execució del xifrador a la GPU es pot veure una més que notable diferència. Si bé abans de fer les proves i tenint en compte el funcionament de la tecnologia CUDA es podria pensar que la GPU trigaria el mateix en tractar un únic *State* que en tractar cent o mil, ja que tots els fils s'executen en paral·lel i al mateix temps. Però, les dades recollides contradiuen aquesta suposició ja que es pot observar que si bé el temps d'execució en tots els casos és molt reduït comparat amb la versió CPU es pot apreciar un creixement del temps d'execució proporcional al creixement de la mida de les dades que ha de tractar. Ja que aquest temps també inclou el temps que es triga en l'intercanvi de dades entre la memòria principal i la memòria de la GPU, el més probable és que aquest increment sigui degut a l'intercanvi de les dades entre la memòria principal i la memòria de la GPU i que la hipòtesi anterior fos encara correcta.

Per a comprovar si aquest increment es provocat per la transferència es pot comprovar fàcilment fent una petita modificació al codi i executant una altra vegada les proves. Només cal comentar la línia on es crida al kernel, d'aquesta manera el xifrador només enviarà i rebrà les dades sense tractar-les a la GPU. Amb els temps resultant, només cal restar-los als temps anterior i es veurà si el temps restant (que és el temps que es triga en tractar les dades) es manté constant o depèn de les dades. Els resultats són els que es poden veure a la taula 7.2.

En contra del que s'esperava, el temps de càlcul creix en proporció a les dades tractades i el temps de transferència sorprèn amb un creixement proporcional invers a les dades tractades.

	Temps de l'intercanvi	Temps de càlcul	Temps total
128 MB	0.79 s	0.53 s	1.32 s
256 MB	0.71 s	0.96 s	1.67 s
384 MB	0.64 s	1.35 s	1.99 s
512 MB	0.63 s	1.68 s	2.31 s

Taula 7.2: Temps de tractament de les dades a la GPU

El motiu d'aquestes dades s'hauria d'investigar a un nivell més baix que el aplicació, ja que aquestes dades poden ser degudes a la implementació dels controladors del dispositiu o bé de la pròpia implementació del hardware. De totes formes, és un estudi independent que no es tracta en aquest projecte.

Independentment dels temps de transferència i dels temps de càlcul, queda demostrat que en el que a temps total respecta, la tecnologia CUDA és molt superior a la versió CPU.

7.3 Consum energètic

Un altre aspecte a tenir en compte en la comparativa és el consum elèctric. Cal tenir en compte que en la GPU hi ha molt més hardware que no pas a la CPU tal i com es pot veure a l'apartat on s'explica l'arquitectura dels dispositius CUDA (veure apartat 5.1.1) i això pot tenir conseqüències en el consum d'energia d'aquests dispositius.

El problema apareix en el moment en que es vol mesurar el consum d'un component de dintre de l'ordinador, no hi ha cap software que ho mostri. Llavors, l'única alternativa que es presenta és la mesura del consum de forma externa a l'ordinador.

El consum elèctric es mesura en KWh (Kilowatts x Hora), i si ja es tenen les dades del temps que triga l'execució de cada versió, només falta saber quina és la potència consumida per la CPU i la GPU mentre es tracten les dades al xifrador. Cal recordar que la potència es pot calcular mitjançant la fòrmula:

$$Potència = Voltatge \cdot Intensitat$$

En aquesta fòrmula hi ha dues incògnites: la potència (variable que es vol calcular) i la intensitat (variable desconeguda), ja que el voltatge és conegut i constant (220 Volts). Llavors, cal mesurar la intensitat de corrent que entra a l'ordinador en el moment del tractament de les dades. Això s'ha fet usant una pinça amperimètrica que es mostra a la fotografia 7.2.

Encara que la potència es calcula multiplicant tensió elèctrica per intensitat, això només és vàlid amb el corrent continu. Si es mesura corrent altern, a aquesta fòrmula cal aplicar-li un factor de correcció anomenat *factor de potència*. No s'explicarà en que consisteix i s'usarà la fòrmula abans descrita ja que dóna una aproximació al valor real i permetrà generar uns valors orientatius.

El muntatge per a realitzar la mesura és senzill. S'ha utilitzat un cable doble entre l'endoll de la paret i el cable connectat mitjançant una regleta al cable de la font d'alimentació de l'ordinador. A aquest cable se li han separat els dos cables que el formen just abans d'arribar a la regleta per tal de fer passar només un d'ells per la pinça amperimètrica. Cal tenir en compte que si es passen els dos cables per la pinça la lectura serà nul·la. El muntatge final queda tal i com es veu a la figura 7.3, on el cable negre connectat a la regleta és el que connecta amb la font d'alimentació de l'ordinador.

Per a calcular només el consum de la CPU i la GPU, s'haurà de restar el consum de tota la resta del hardware de l'ordinador. Per fer això, cal fer tres mesures: ordinador en repòs, tractament de dades amb la CPU i tractament de dades amb la GPU. Fent aquestes mesures

Figura 7.2: Amperímetre usat per mesurar la intensitat del corrent

els resultats són els que es veuen a la taula 7.3. Cal recordar que les mesures del Core i7 estan fetes a un altre ordinador per això les intensitats en repòs són tant diferents, l'equip en qüestió té molt més hardware (discos durs) i la placa base també és diferent.

	Repòs C2D	Execució C2D	Repòs i7	Execució i7	Execució GPU
Intensitat	0.52 A	0.57 A	0.9778 A	1.044 A	0.947 A

Taula 7.3: Intensitats mesurades

Per a veure la potència consumida pels components que es volen comprar cal restar-li la intensitat mesurada amb l'ordinador en repòs i multiplicar el resultat pel voltatge de la línia, que és de 220 volts. Aquests càlculs donen les potències que es poden veure a la taula 7.4 seguint la fòrmula:

$$220\,V \cdot \mathit{Intensitat} = \mathit{Potència}$$

	C2D	i7	GPU
Potència	11 W	14.56 W	93.94 W

Taula 7.4: Potències durant el tractament de les dades

Es pot veure als valors obtinguts una gran diferència de potència consumida entre els casos en que s'usa la CPU i el cas en que s'usa la GPU. Tal i com es sospitava, la GPU consumeix molta més energia que no pas les CPU. I una altra fet que resulta curiós és el menor consum del C2D respecte el i7. Ara cal veure la combinació d'aquests valors amb els temps de tractament de les dades de cada dispositiu i veure quin consumeix més energia.

Per poder calcular el consum en KWh del tractament de les dades en cada cas juntament amb els temps d'execució mesurats anteriorment cal fer ús de la formula:

Figura 7.3: Muntatge per mesurar la intensitat del corrent a la font d'alimentació de l'ordinador

$$Potència\,(W)\,\cdot\,\frac{1\,KW}{1000\,W}\cdot\,Temps\,(s)\,\cdot\,\frac{1\,H}{3600\,s}=Consum\,(KWh)$$

Amb aquests càlculs, els consums resultants per a cada cas són els que es poden veure a la taula 7.5.

	E6600	i7	GPU
128 MB	0,000235736 KWh	0,000232085 KWh	0,000034445 KWh
256 MB	0,000480761 KWh	0,000464452 KWh	0,000043578 KWh
284 MB	0,000733517 KWh	0,000696846 KWh	0,000051928 KWh
512 MB	0,000977778 KWh	0,000929128 KWh	0,000060278 KWh

Taula 7.5: Consums d'energia pel tractament de les dades

7.4 Cost econòmic

Com es pot veure a la figura 7.5, en termes absoluts, la GPU consumeix molta menys energia que les dues CPU amb les quals es compara. Doncs, en consum d'energia, la GPU consumeix menys que les altres dues CPU. Per a veure aquestes dades d'una forma que tothom ho entengui, s'expressarà aquesta energia amb el seu valor al mercat, és a dir, quants diners costa el tractament d'aquestes dades amb cada dispositiu? Doncs, només cal aplicar la següent fòrmula:

$$Energia\,\cdot\,\frac{preu}{KWh}=Cost$$

El preu que s'ha aplicat és la tarifa TUR[24], la Tarifa d'Últim Recurs que el govern central fixa pels usuaris, que actualment és de 0.142319 €/KWh. Aplicant aquesta tarifa, surten els

costos de la gràfica 7.4, on el cost està representat en cèntims d'Euro per a facilitar la seva lectura.

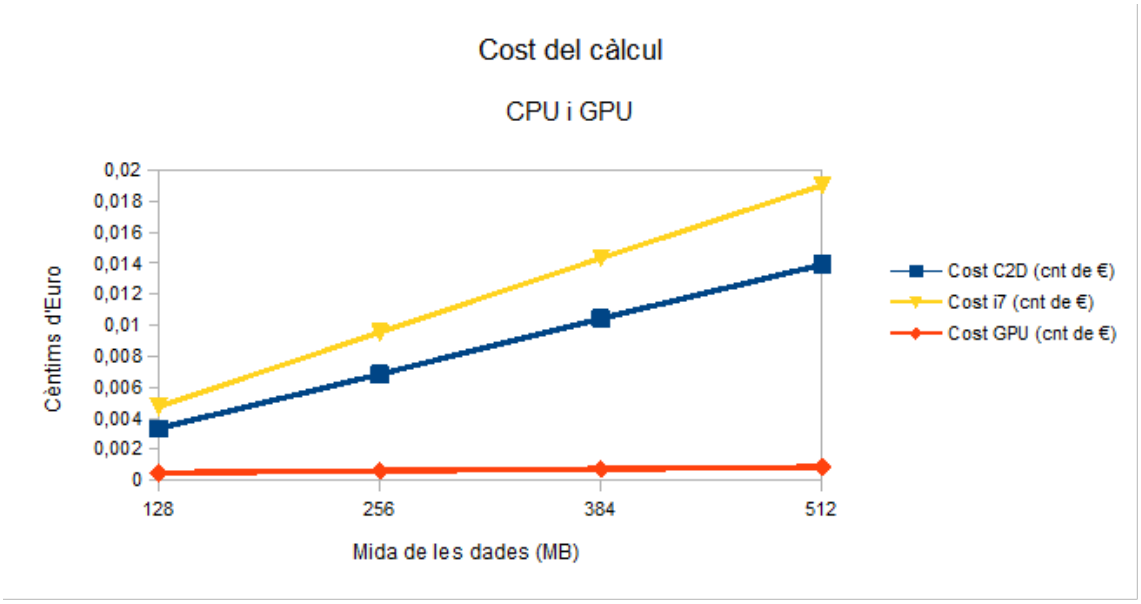

Figura 7.4: Cost econòmic pel tractament de les dades

Es pot veure com la diferència en les corbes de creixement és notable entre les CPU i la GPU encara que és una gràfica que es mou en xifres econòmiques molt petites. En el cas de les CPU el creixement en ambdós casos és proporcional a les dades que han de tractar, però en el cas de la GPU, el creixement és molt més suau i gairebé no s'aprecia el creixement de la corba.

Per tant costa molt menys executar el xifrador AES mitjançant hardware CUDA que amb la CPU, encara que si el software estigués optimitzat per aprofitar tots els nuclis de les CPU provades (2 fils d'execució en el cas del E6600 i fins a 8 fils d'execució en el cas del i7 920) aquesta gràfica es veuria afectada.

Per altra banda, la CPU és un element necessari per al normal funcionament de l'ordinador. Però en canvi, la GPU és un dispositiu opcional, per tant presenta un sobre-cost. Per a poder executar la versió GPU del xifrador es necessita del hardware específic, que en aquest cas la GPU utilitzada per a aquest projecte té un preu actual al mercat d'uns 120€ aproximadament. Doncs, inicialment, executar el xifrador suposa 120€ més el cost de cada execució. Per tant, amb el xifrador AES, quantes vegades s'han de calcular 512MB de dades per a amortitzar el sobre-cost de la compra de la GPU?

Si aquest càlcul ho fem respecte el model Core 2 Duo E6600, s'ha de realitzar la següent equació:

$$Cost\,CPU = Cost\,GPU$$

$$X \cdot Cost\,CPU\,512\,MB = 120€ + X \cdot Cost\,GPU\,512\,MB$$

$$X \cdot 0.01391563 = 120€ + X \cdot 0.00085787$$

Aïllant la X, trobem:

$$X = \frac{120€}{0.01305776} = 9189.937$$

Aquest és el nombre d'unitats de 512MB que cal tractar per a amortitzar el sobre-cost de la GPU, que en unitats de volum de dades és:

$$512\,MB \cdot 9189.937 = 4705248,067049\,MB \cdot \frac{1\,GB}{1024\,MB} \cdot \frac{1\,TB}{1024\,GB} = 4.487\,TB$$

Per una altra banda, fent el càlcul és en unitats de temps sense tenir en compte el temps de llegir i escriure en disc les dades, només el temps que triga la GPU en tractar els 512MB és:

$$2.31\,s \cdot 9189.937 = 21228.7559\,s \cdot \frac{1\,h}{3600\,s} = 5.8968\,hores$$

És a dir, caldria processar 4.487TB d'informació o processar informació durant 5.8969 hores de forma ininterrompuda (i sense tenir en compte el temps d'accés a disc). Doncs, aquest sobre-cost no és gaire important tenint en compte que en un sol es pot amortitzar el cost del hardware usat en les proves d'aquest projecte.

Cal recordar que tant els consums com les estimacions de costos són orientatius (a causa de no tenir en compte el factor de potència en el càlcul de la potència consumida per l'ordinador) i el més probable és que els costos i l'amortització siguin majors, tot i així les proporcions seran les mateixes a les presentades en aquest projecte.

7.5 Temperatures

Ara ja s'han vist els temps que triguen les CPU i la GPU en tractar les dades i quina quantitat d'energia consumeixen. Però, es planteja la següent pregunta: Si la GPU consumeix més energia, s'escalfarà molt més que les CPU? S'ha de tenir en compte aquest aspecte, perquè en entorns industrials, si per exemple es munta un Centre de Computació, s'haurà de tenir en compte el cost de refrigerar aquests equips.

Per resoldre aquest dubte s'ha fet us del software Xsensors, una aplicació que dóna molta informació sobre els diferents sensors repartits per tot l'ordinador, entre ells la temperatura de la CPU. Per altra banda per saber la temperatura de la GPU s'ha fet servir el propi controlador de NVIDIA, qui té una secció on mostra la temperatura de la GPU en temps real. Es pot veure un exemple d'aquests software a la imatge 7.5, on es veu les temperatures de la CPU i la GPU mentre l'ordinador està en repòs.

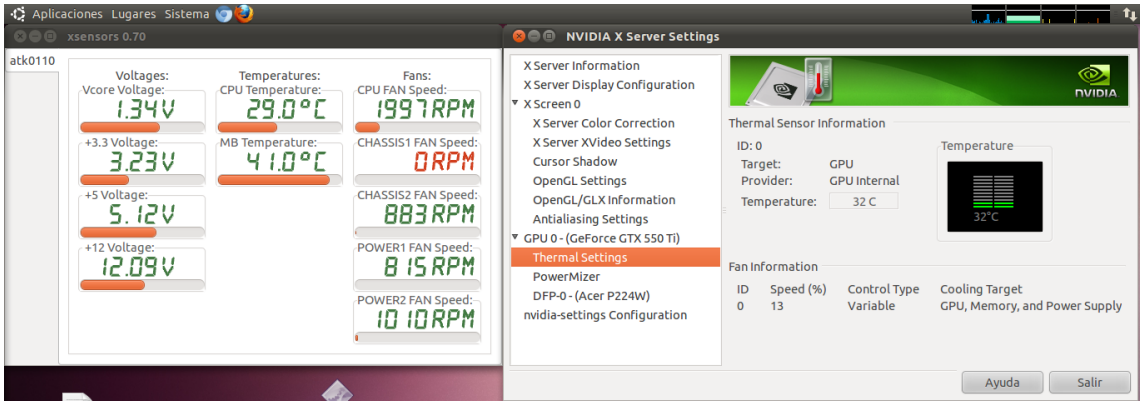

Figura 7.5: Temperatures de la CPU i la GPU mentre l'ordinador està en repòs

Doncs, per veure les diferències es prenen tres mesures de les temperatures, en els mateixos casos que la de la intensitat: ordinador en repòs, fent us de la CPU, i fent us de la GPU. Però, no s'han mesurat les temperatures de la CPU i7 ja que amb les dades del E6600 són suficients

	C2D	GPU
Repòs	29ºC	32ºC
Usant CPU	40ºC	32ºC
Usant GPU	30ºC	46ºC

Taula 7.6: Temperatures dels components

per a tenir una idea aproximada de la diferència de temperatures. Les dades recollides usant aquest software han estat les que es mostren a la taula 7.6.

Es pot apreciar que la diferència de temperatures en ambdós casos és molt semblant. En el cas de la CPU, la diferència es d'uns 10ºC aproximadament, mentre que en el cas de la GPU la diferència tèrmica creix fins a 12ºC. Però cal tenir en compte que en el cas de la GPU, aquesta treballa durant un curt període de temps (entorn als 2 segons) amb el xifrador mentre que la CPU manté aquestes temperatures durant tot el temps que triga en tractar les dades. Caldria executar algun algorisme a la GPU que trigués més temps per a veure si aquests 46ºC que marca en el cas del xifrador AES es mantenen o, en canvi, aquesta temperatura segueix augmentant fins a una cota superior a aquest valor.

Es pot dir, doncs, que en el cas d'usar intensivament la GPU, aquesta produeix més calor que no pas la CPU. Per tant, un centre de computació consumirà més energia en refrigerar les GPU que no pas en fer-ho per a les CPU.

Com es pot veure amb les dades dels apartats 7.3 i 7.5, la plataforma CUDA és una tecnologia més ecològica que la CPU en les tasques analitzades. S'ha vist com aquesta tecnologia amortitza el seu cost inicial molt ràpidament i consumeix menys electricitat durant el seu funcionament, cosa que redueix l'emissió de CO2 a l'atmosfera per la generació de l'electricitat.

Capítol 8

Planificació final i costos

8.1 Planificació final

En el tercer capítol d'aquest projecte s'ha especificat una planificació de les tasques a realitzar assignant uns dies específics i unes determinades hores per a la seva realització, però durant la realització del projecte aquestes dates no s'han pogut complir.

El principal motiu ha estat la implementació de la versió CUDA de la biblioteca libvpx ja que ha donat més problemes dels esperats. Inicialment es va cometre l'error de no generar una configuració del codi font íntegrament en C, i el desconeixement de que això estava provocat pel configurador va portar a perdre temps cercant un paral·lelisme que no es pot donar ja que la descodificació del fotograma implicava aquest codi en assemblador.

És per això que la data de finalització del projecte s'ha endarrerit un total de 10 dies. A la figura 8.1 es pot veure com ha quedat la planificació final del projecte, es pot apreciar l'augment dels dies assignats a la implementació de les aplicacions.

	Modo de tarea	Nombre de tarea	Duración	Comienzo	Fin
1		**PFC**	**112,75 días**	**jue 01/09/11**	**dom 22/01/12**
2		**Recopilació de tecnologies**	**22 días**	**jue 01/09/11**	**mié 28/09/11**
3		Análisi de WebM (libvpx)	10 días	jue 01/09/11	mar 13/09/11
4		Análisi de CUDA	10 días	mar 13/09/11	lun 26/09/11
5		Análisi de la GUI	2 días	lun 26/09/11	mié 28/09/11
6		Documentació	4 días	jue 01/09/11	lun 05/09/11
7		**Implementació**	**94,75 días**	**jue 01/09/11**	**vie 30/12/11**
8		Implementació de l'aplicacions per consola	60 días	mié 05/10/11	mar 20/12/11
9		Testeig i correcció de les aplicacions per consola	8 días	mar 20/12/11	vie 30/12/11
10		Documentació	5 días	jue 01/09/11	mié 07/09/11
11		**Redacció de la memòria**	**18 días**	**vie 30/12/11**	**dom 22/01/12**
12		Analisi del rendiment de les versions CUDA	8 días	vie 30/12/11	lun 09/01/12
13		Compilació de tota la documentació	10 días	lun 09/01/12	dom 22/01/12

Figura 8.1: Planificació final del projecte

8.2 Cost del projecte

Un altre aspecte en la realització d'un projecte com el present a banda de la seva planificació és el cost que té aquest projecte, i per fer-ho cal tenir en compte els costos de material i mà d'obra. Ja que el hardware CUDA usat ja es trobava en possessió de l'autor i que els instruments i

material per a fer les mesures de consum elèctric han estat cedits per un familiar, només es tindrà en compte el cost del temps invertit en la realització del projecte.

Per a calcular aquest cost cal tenir en compte que l'horari realitzat ha estat el que s'indica a la planificació inicial del projecte (veure 4.3), i és el següent:

- De dilluns a divendres de 16.00 hores fins a les 22.00 hores.
- Caps de setmana de 10.00 a 13.00 i de 15.00 a 19.00 hores.

Això suposa un total de 44 hores a la setmana. Però, també cal tenir en compte que durant els dies compresos per a la realització del projecte hi ha hagut dies festius, i dies en els que no s'ha dedicat cap hora per a treballar en el projecte. En total han estat 8 dies, que són:

- Desembre: dies 16,17,18, 25, 26 i 31.
- Gener: dies 1 i 6.

Tenint en compte que entre la data d'inici (1 de Setembre de 2011) i la data de finalització (22/01/2012) hi ha un total de 144 dies, als qual s'han de restar els dies no treballats. Llavors, ja es pot calcular el nombre total de dies treballats, encara que per facilitar els càlculs posteriors també s'expressa en setmanes.

$$144 - 8 = 136\,dies \cdot \frac{1\,setmana}{7\,dies} = 19,42\,setmanes$$

Una vegada ja tenim el nombre d'hores per setmana i el nombre de setmanes treballades, el nombre d'hores total treballades és de:

$$19.42\,\frac{h}{setmana} \cdot 44\,setmanes = 854,85\,hores$$

Ara, cal donar un valor econòmic a aquestes hores. Si al client se li cobrés cada hora a un preu fictici de 40 €/h, llavors, el cost de la mà d'obra per a aquest projecte és d'un total de:

$$854.85\,hores \cdot 40\,\frac{€}{h} = 34194\,€$$

Capítol 9

Conclusions i treball futur

9.1 Conclusions

Les conclusions que es poden treure de tot el treball fet en aquest projecte són vàries, però es poden classificar en dos grans blocs: el desenvolupament de les aplicacions amb tecnologia CUDA, i el rendiment d'aquesta durant l'execució del programa respecte la versió CPU del mateix.

Durant els darrers anys el hardware de les CPU ha apostat pel paral·lelisme per a millorar el rendiment en l'execució dels programes. Però, les CPU destinades al mercat domèstic no estan pensades per a una computació massiva de dades, és a dir, no estan dissenyades per estar contínuament fent càlculs amb un gran volum de dades. Pel contrari, les GPU si que estan pensades per a realitzar aquesta tasca ja la feina de renderitzar gràfics requereix d'una gran potència de càlcul. S'ha vist com la plataforma CUDA de NVIDIA permet aprofitar aquest gran potencial per a fer tasques més genèriques que no tenen res a veure amb la representació de gràfics.

En el desenvolupament d'aplicacions, es pot veure que l'adaptació d'un programa escrit en C/C++ a CUDA C/C++ és relativament senzill. No cal canviar gaire el disseny del software ni la implementació d'aquest si la versió original ja està preparada per a dividir el problema en trossos independents. Però, si l'algorisme original és seqüencial, la seva paral·lelització no és possible com ja s'ha vist amb l'exemple de la biblioteca libvpx.

Per a adaptar el codi font original a CUDA C/C++ només cal redefinir els objectes que s'usen i les interfícies de les funcions per a que el compilador CUDA entengui que són kernels. Si bé aquesta part és senzilla, potser la part més complexa de tota la transformació del codi original és la configuració de la crida al kernel. Però, en contrapartida, la depuració del codi font dels kernels no es pot fer de la mateixa manera que amb el codi C/C++ original.

En el cas del software per a CPU, la depuració del codi és una tasca fàcil. Existeixen depuradors de codi que mostren l'estat de l'algorisme en l'execució de cada línia, encara que la depuració sempre es pot fer des del mateix codi fent que aquest imprimeixi missatges de depuració per pantalla durant la seva execució. Però en el cas de CUDA, només existeix un depurador de codi anomenat CUDA Parallel Nsight que és un software propietari de NVIDIA i que es troba només disponible per a Windows encara que és un software gratuït. Això fa que la depuració sigui una tasca molt més difícil de realitzar en el cas del desenvolupament de software per a CUDA.

Tenint el compte el software del xifrador AES, la versió CUDA no ha costat gaire d'implementar. Per una altre banda, en el cas de les modificacions de la biblioteca libvpx s'ha pogut veure com la seva transformació a CUDA C/C++ no és pas trivial ja que seguint el mateix procés que al cas del xifrador no s'ha pogut arribar a una versió completament funcional i a més a més s'ha hagut de redefinir molt més codi.

Per tant, es pot afirmar que des del punt de vista del desenvolupador, la transformació d'una aplicació a CUDA C/C++ és una tasca d'una complexitat mitjana ja que depèn de la complexitat del codi font original per a CPU.

Deixant de banda el llenguatge de programació, durant l'anàlisi del codi original de la biblioteca libvpx s'han vist elements software propis del llenguatge C abans desconeguts, com és el cas de les macros definides com a funcions i s'ha après, sense profunditzar gaire, com funciona un còdec de vídeo. A més a més, s'ha après a usar una biblioteca estàtica per a la encapsulació de part del codi i incloure-la a la resta del software i com fer que el software desenvolupat pugui ser compatible amb arxius de gran volum.

Respecte al funcionament del software CUDA desenvolupat, ha sorprès la gran superioritat de la potència de càlcul de la GPU sobre la CPU. Aquesta superioritat es manifesta clarament en el temps d'execució, aspecte que fa que el consum energètic pel tractament d'una quantitat limitada de dades sigui més reduït en el cas d'usar la gran paral·lelització que proporciona la plataforma CUDA. Encara que existeix un cost inicial elevat, per a adquirir el hardware necessari, la quantitat d'energia que consumeix aquest executant el software escrit en CUDA C/C++ fa que la seva amortització sigui ràpida.

Doncs, amb totes les conclusions explicades anteriorment, es pot afirmar que la plataforma CUDA per a l'acceleració de processos és totalment viable només si l'algorisme original facilita el seu paral·lelisme.

L'objectiu d'aquest projecte era analitzar la viabilitat de la tecnologia CUDA per accelerar processos. Però, per a a arribar a assolir aquest objectiu ha calgut assolir una sèrie d'objectius al començament del projecte. Aquests objectius són els llistats a l'apartat 1.2. A continuació es descriu si aquests objectius s'han pogut arribar a complir durant el desenvolupament del projecte.

- Estudiar la paral·lelització dels algorismes de la biblioteca libvpx i el xifrador AES
  Aquest objectiu s'ha pogut assolir en la seva totalitat. Tal i com s'explica al capítol cinquè, amb aquest estudi s'ha pogut concloure que l'algorisme de la biblioteca libvpx no és totalment paral·lelitzable ja que gran part de l'algorisme té un disseny seqüencial, per altra banda el xifrador AES és totalment paral·lelitzable.

- Crear aplicacions CUDA per a la biblioteca libvpx i el xifrador AES
  Aquest objectiu no s'ha pogut arribar a complir ja que el desenvolupament de la versió CUDA de libvpx no ha arribat a produir un software completament funcional. Per altra banda, en cas del xifrador AES s'ha pogut desenvolupar una versió CUDA totalment funcional i que ha permès assolir d'altres objectius plantejats.

- Analitzar el consum d'energia i temps d'execució de les versions originals i les versions CUDA de la biblioteca libvpx i el xifrador AES
  Aquest objectiu depèn de l'anterior, i com no s'ha pogut desenvolupar la versió CUDA de la biblioteca libvpx, no s'ha pogut fer l'anàlisi del consum d'energia ni el temps d'execució. En canvi, si que s'ha pogut fer pel xifrador AES.

- Estudi dels costos de hardware i del consum energètic
  En el capítol setè s'ha fet un estudi del sobre-cost del hardware i consum energètic del software CUDA per a analitzar si el seu consum és més reduït que la versió per a CPU del mateix software i quin és el temps que es triga en amortitzar el sobre-cost que suposa la compra del hardware necessari. Per tant, aquest objectiu s'ha assolit sense problemes.

- Concloure la viabilitat de la tecnologia CUDA per a l'acceleració de processos
  Aquest és l'objectiu principal del projecte que s'ha pogut assolir gràcies a l'estudi dels costos de l'objectiu anterior. Tal i com s'explica en el capítol set, la tecnologia CUDA és viable segons el nivell de paral·lelisme del software en qüestió.

9.2 Treball futur

L'assoliment del principal objectiu d'aquest projecte indica la viabilitat d'aquesta tecnologia per accelerar processos. I tal i com s'ha intentat fer amb la biblioteca libvpx es pot migrar diferents algorismes de compressió de vídeo a la tecnologia CUDA, ja que molts utilitzen l'estructura de macroblocs per al tractament dels fotogrames. Però els compressors de vídeo no sons els únics que utilitzen aquesta estructura, la compressió d'imatges JPEG també fa ús de la divisió de la imatge en macroblocs. Tanmateix, existeixen molts algorismes paral·lelitzables, per exemple es podria fer una versió en aquesta plataforma del compressor d'arxius BZIP2, que divideix la compressió en blocs.

Seguint amb el desenvolupament d'aplicacions per a CUDA, en aquest projecte s'ha desenvolupat una versió del xifrador AES completament funcional que funciona amb el hardware on s'ha realitzat el seu desenvolupament. Però si aquesta aplicació es vol executar en altres entorns potser que no funcioni. Per tant, es per a treball futur fer la versió del xifrador compatible amb d'altres GPU amb una mida de memòria o capacitat de càlcul diferent. A més a més, existeix la possibilitat d'estudiar l'optimització del codi font del xifrador usant les diferents memòries disponibles a la GPU i optimitzar la configuració de la crida al kernel. Pel que fa a la versió CPU es pot fer una versió que aprofiti tots els nuclis i fils d'execució del processador central. D'aquesta manera es podria aprofitar tot el potencial de la CPU i veure si encara segueix sent viable la tecnologia CUDA, encara que tots els indicis no indiquen el contrari.

Pel que fa a la versió CUDA de libvpx, ha de depurar-se el seu codi font i veure quina és la causa que provoca que el vídeo resultant no sigui l'esperat. Aquesta tasca serà difícil, ja que la depuració del codi CUDA es troba molt lluny de ser trivial. Però, la comunitat de desenvolupadors del projecte WebM ha ajudat a l'autor durant amb les modificacions fetes al codi en aquest projecte explicant tot allò que calia i aportant solucions als problemes presentats. Una vegada s'hagi depurat el codi i funcioni correctament, es podrà optimitzar assignant un fil d'execució de la GPU a cada macrobloc del fotograma sense haver d'iterar per cada fila de macroblocs.

Finalment, ja en l'estudi del rendiment de l'aplicació CUDA s'ha vist un comportament inesperat envers els temps d'execució del kernel i el temps de l'intercanvi de dades entre la memòria de la GPU i la memòria principal del sistema. Això obre una nova via de investigació per a trobar el motiu que provoca aquest comportament.

Apèndix A

Preparació del entorn de desenvolupament

A totes dues plataformes (Windows 7 i Ubuntu 10.10) cal descarregar els següents elements, que es troben disponible a la zona per a desenvolupadors de la pàgina web de NVIDIA[3], per a poder desenvolupar software que usi tecnologia CUDA:

- Controlador de hardware per a desenvolupadors
- *CUDA Toolkit*
- *GPU Computing SDK*

Cal dir que durant les instal·lacions que s'expliquen a continuació, les rutes usades són les que els instal·ladors proposen per defecte tant a Windows 7 com a Ubuntu 10.10.

A.1 Windows 7

Primerament, s'ha d'instal·lar el controlador del hardware per a desenvolupadors. Aquesta instal·lació es realitza mitjançant el típic assistent d'instal·lació de Windows. Però, per a que no hi hagi problemes amb controladors anteriors, cal indicar que es faci un intal·lació correcta mitjançant l'instal·lació personalitzada tal i com es mostra la figura A.1.

Una vegada instal·lat el controlador, cal instal·lar el *CUDA Toolkit* i el *GPU Computing SDK*. A més a més, cal instal·lar l'IDE Microsoft Visual C++ 2008, disponible per a descàrrega de forma gratuïta a la web de Microsoft[14]. Els instal·ladors són molt senzills d'usar i per això no s'expliquen, i en el cas del *GPU Computing SDK* ja s'instal·len tots els exemples de forma automàtica.

Després d'instal·lar tot el software anterior, cal configurar l'IDE per a desenvolupar una aplicació CUDA. Encara que el desenvolupament es fa usant el llenguatge CUDA C/C++, cal crear un projecte buit amb el menú *File->New->Project* i escollint l'opció *Empty Project* tal i com es mostra a la figura A.2.

Una vegada creat el projecte cal configurar-lo per a que usi el compilador de CUDA C/C++. Aquesta configuració consta de diferents passos, el primer és assignar les regles de compilació al projecte. Aquestes regles de compilació es troben al directori *C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v4.0\extras\visual_studio_integration\rules.*

Per carregar-les dintre Visual C++, només cal accedir al menú que es mostra a la figura A.11 i afegir-les a la llista de regles amb el botó *Find Existing...* que es mostra a la figura A.4, una vegada fet això cal activar-les marcant la casella corresponent de la llista tal i com es mostra a la mateixa imatge.

Una vegada configurades les regles de compilació, cal configurar l'enllaçador. Per fer-ho, s'accedeix a les seves opcions a través de l'opció de propietats del projecte com es mostra a la figura A.5.

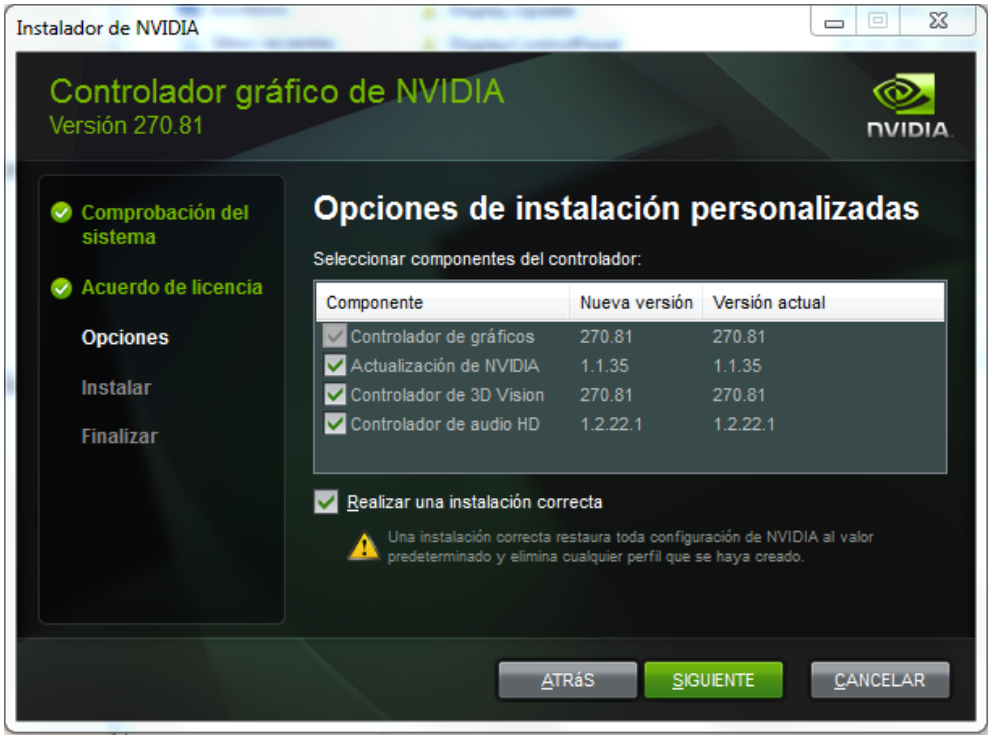

Figura A.1: Instal·lació del controlador mitjançant una instal·lació correcta

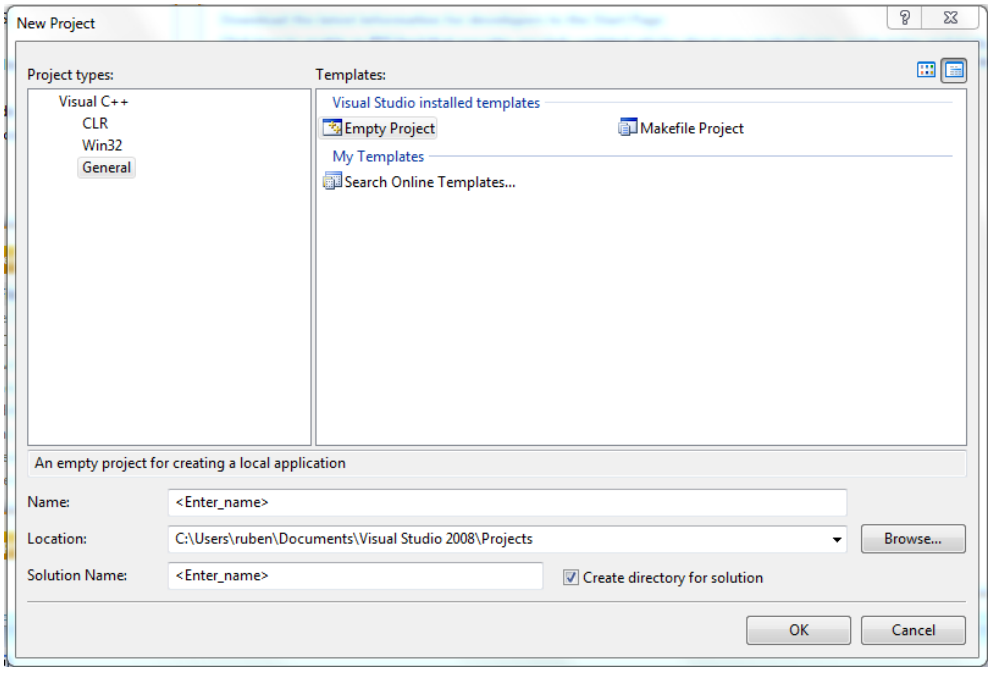

Figura A.2: Creació d'un projecte buit en Visual C++ 2008 Express

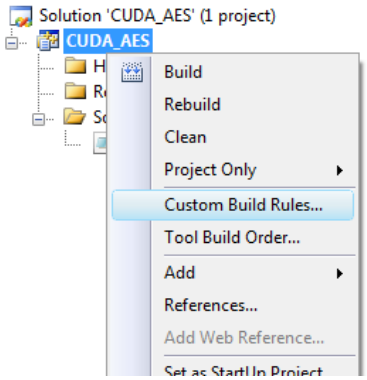

Figura A.3: Opció per a afegir les regles de compilació de CUDA

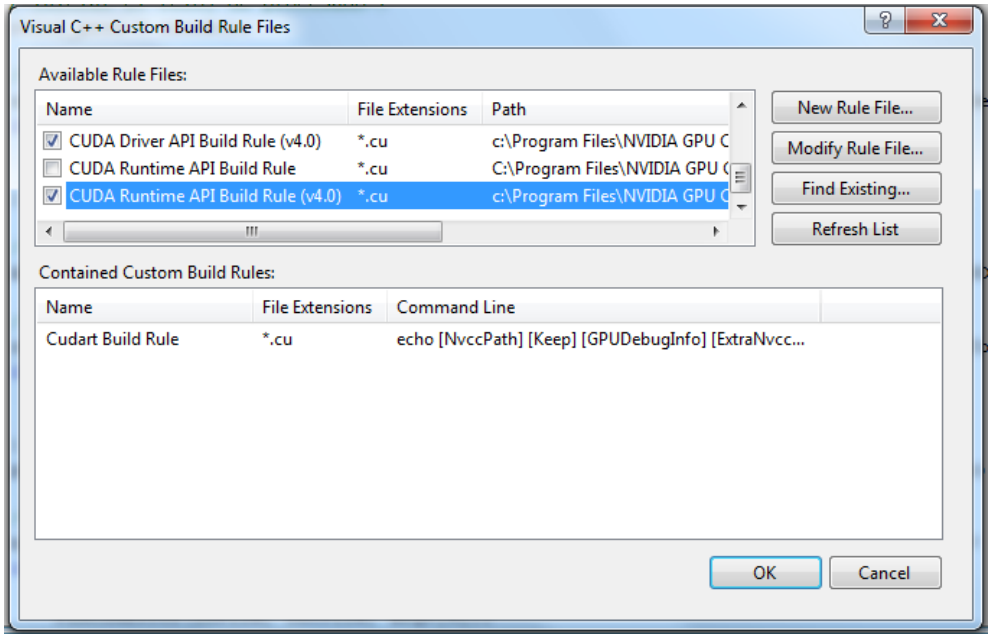

Figura A.4: Llista de regles de compilació del projecte

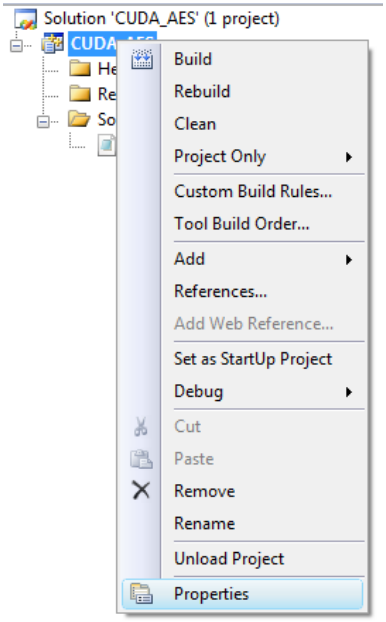

Figura A.5: Opció per accedir a les propietats del projecte.

Una vegada a la finestra de propietats del projecte el primer que cal fer es accedir a les opcions generals de l'enllaçador i modificar el valor del camp *Additional Library Directories* pel valor que es mostra a la figura A.6 que és la variable global que crea l'instal·lació del SDK per indicar la seva pròpia ruta. Després, cal accedir a les opcions d'entrada i modificar el camp *Additional Dependencies* tal i com es mostra a la figura A.7. A més a més, en aquesta finestra també es pot trobar el camp on posar els arguments d'execució per a quan s'executi el programa a través de Visual C++, és a l'apartat *Debugging* dintre de *Configurations Properties*.

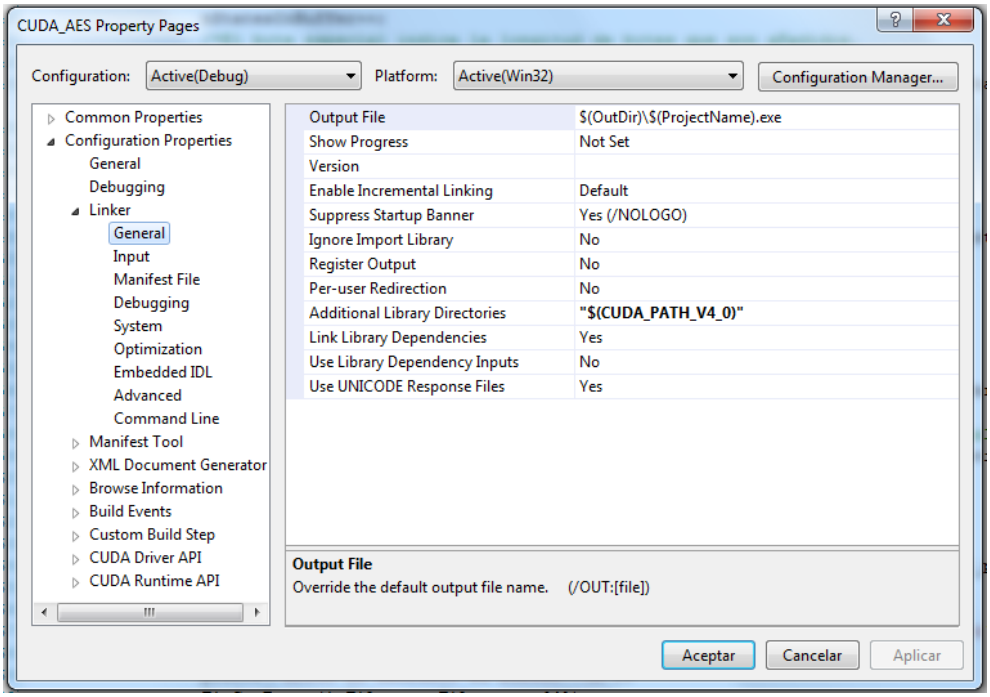

Figura A.6: Opcions generals de l'enllaçador

El següent pas és configurar certes configuracions generals de Visual C++ 2008. Per fer-ho, cal accedir-hi mitjançant el menú general *Tools->Options*. A la finestra que apareix, cal desplegar el menú *Projects and Solutions* de la part esquerra tal i llavors apareixen les opcions que es poden veure a la figura A.8.

En aquesta finestra cal afegir les rutes de: la biblioteca CUDA, els executables de la biblioteca, i els fitxers d'inclusions. Per afegir tots aquests paràmetres cal canviar el desplegable *Show directories for:* i afegir les següents rutes:

- A l'opció *Executable files* cal afegir la ruta *$(CUDA_PATH)bin*
- A l'opció *Include files* cal afegir la ruta *$(CUDA_PATH)include*
- A l'opció *Library files* cal afegir la ruta *$(CUDA_PATH)lib\Win32*

Per últim, dintre de les opcions generals de Visual C++, per tal que aquest reconegui les extensions *.cu* i *.cuh*, cal afegir-les al camp *C/C++ File Extensions* de l'opció *VC++ Project Settings*, tal i com es mostra a la figura A.9.

Per últim, només cal assegurar-se que el codi font sigui compilat pel compilador de CUDA C/C++. Per a comprovar-ho, cal accedir a les propietats de cada fitxer *.cu* del codi font i comprovar que a les opcions generals, l'eina de compilació escollida és *CUDA Runtime API*, tal i com es veu a la figura A.10.

Ara que ja està tot l'entorn de desenvolupament configurat es pot començar el desenvolupament de l'aplicació CUDA, i quan es vulgui compilar el codi font es selecciona l'opció *Build* del menú contextual del projecte tal i com mostra la figura A.11.

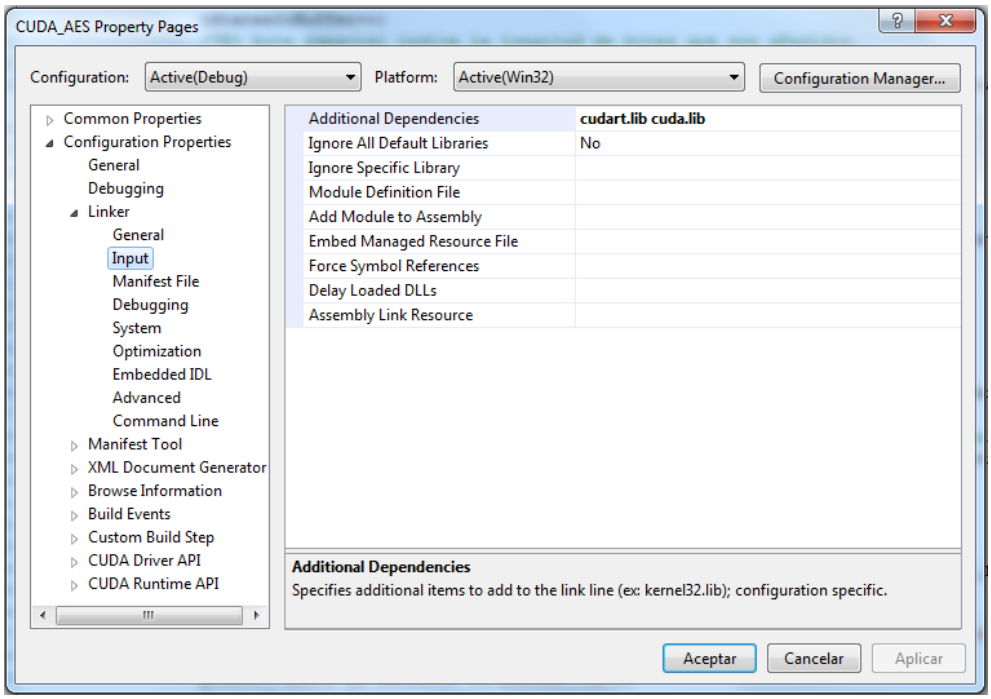

Figura A.7: Opcions d'entrada de l'enllaçador

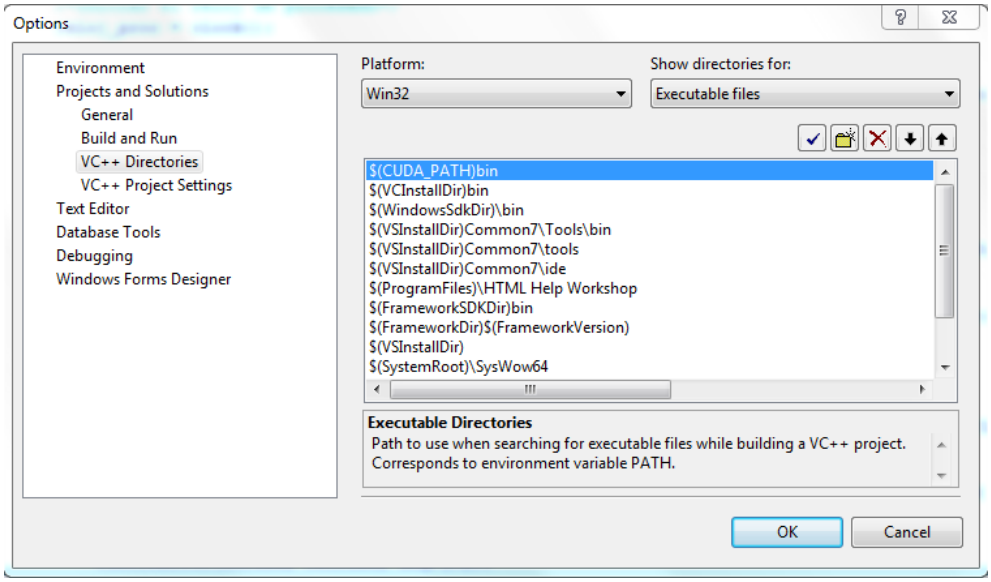

Figura A.8: Finestra d'opcions generals per a projectes i solucions

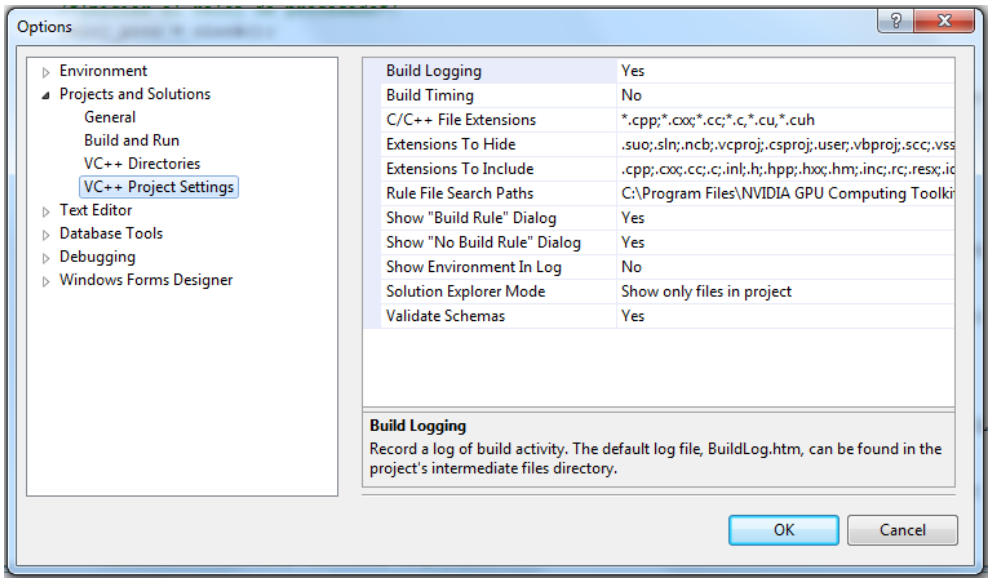

Figura A.9: Opcions generals de projecte

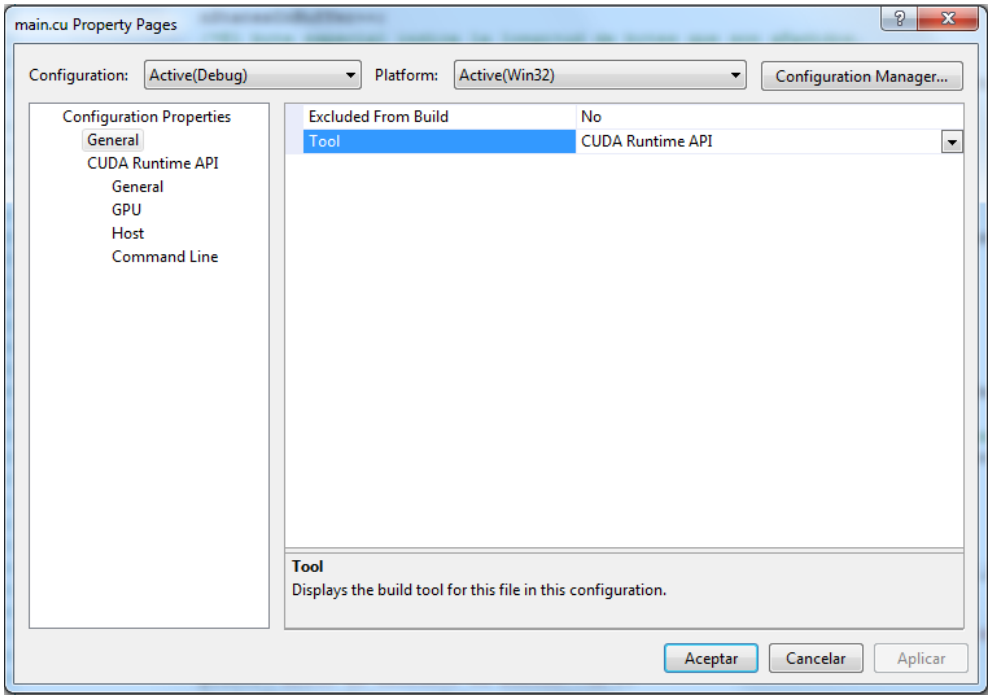

Figura A.10: Propietats generals dels arxiu de codi font

Igual que a qualsevol altre projecte, l'opció *Build* compila el codi font, l'opció *Rebuild* esborra tots els objectes resultants de la compilació i torna a compilar el codi font, i l'opció *Clean* esborra tots els fitxers resultants de la compilació. Això va bé saber-ho ja que compilar amb l'opció *Build* només es compila el codi font modificat i a l'hora de crear l'aplicació pot donar cert problemes al vincular amb les biblioteques.

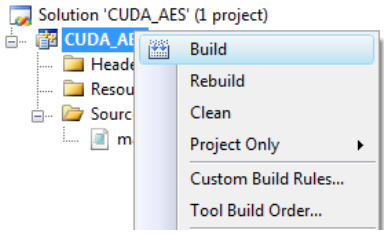

Figura A.11: Opció per a la compilació del codi font

A.2 Ubuntu 10.10

Per a Ubuntu 10.10, els instal·ladors no són fets amb assistents com els de Windows, cal fer ús de la consola de comandes per a realitzar la seva instal·lació.

La instal·lació més complicada és la del controlador del hardware, que també és la primera que s'ha de fer. Per realitzar aquesta instal·lació, s'ha d'iniciar Ubuntu en mode consola, sense l'entorn gràfic. Això es pot fer escollint l'opció *recovery mode* des del GRUB . Una vegada iniciada la sessió en mode text, cal navegar per la consola fins el directori on es troba l'instal·lador i executar-lo amb la comanda: *./devdriver_4.0_linux_32_270.41.19.run*. Això iniciarà un assistent de text per a consola que guia l'instal·lació del controlador.

Cal tenir en compte que aquest instal·lador crea un mòdul per al kernel que s'està executant en el moment de l'instal·lació. Llavors, si s'actualitza el kernel del sistema caldrà tornar a fer aquesta instal·lació si es vol executar Ubuntu amb l'última versió del nucli.

Una vegada ja s'ha instal·lat el controlador de la GPU al kernel, s'han d'instal·lar el *GPU Computing SDK* i el *CUDA Toolkit*. Per instal·lar aquests components no cal iniciar el sistema en mode text, però sí cal usar la consola de comandes. Per executar aquests instal·ladors es navega amb la consola de comandes fins al directori on es troben els intal·ladors i s'executen amb les següents comandes: `./cudatoolkit_4.0.17_linux_32_ubuntu10.10.run` i `./gpucomputingsdk_4.0.17_linux.run`.

Una vegada instal·lats els darrers components, si es vol fer ús d'algun dels exemples continguts al SDK, primer cal compilar-los ja que al contrari que a Windows 7, l'instal·lador no genera els executables dels exemples. Per compilar-los només cal navegar fins al subdirectori *C* que hi ha dins del directori d'instal·lació del SDK amb la consola de comandes i executar la comanda `make`.

Amb tot el que s'ha explicat anteriorment, ja només cal configurar Netbeans 6.9, l'IDE usat per desenvolupar aquest projecte. Per instal·lar-lo només cal accedir al Centre de Software d'Ubuntu des del menú d'aplicacions, cercar-lo i instal·lar-lo. La figura A.12 mostra la finestra del Centre de Software d'Ubuntu una vegada s'hi ha cercat i instal·lat Netbeans 6.9.

Una vega instal·lat l'IDE, cal configurar-lo. Primer de tot cal instal·lar la compatibilitat amb C/C++. Netbeans és un IDE capaç d'interpretar molts llenguatges, però la versió instal·lada des del Centre de Software només té activat el suport per al llenguatge Java. Però, Netbeans disposa d'un gestor d'afegits o *plugins* que al instal·lar-los afegeix noves funcionalitats al IDE. Doncs, cal instal·lar el *plugin* de C/C++ a Netbeans. Per fer-ho, cal accedir al gestor de *plugins* des del menú principal *Tools->Plugins,* i a la finestra que apareix seleccionar el plugin

de C/C++ que apareix a la llista *Available Plugins* i clicar al botó *Install* tal i com es pot veure a la figura A.13.

Una vegada que Netbeans ja té habilitat el suport per a C/C++, cal configurar-lo. Com que en el desenvolupament CUDA s'utilitza un compilador específic, cal dir-li a Netbeans. Ara, cal entrar a les opcions globals de C/C++ a través del menú principal *Tools->Options* i clicar la icona *C/C++*. A la pestanya *Build Tools*, cal duplicar la col·lecció d'eines *GNU* que apareix a la llista esquerra, donar-li el nom de *NVCC* i modificar els camps per tal que quedin tal i com mostra la figura A.14.

Sense marxar d'aquesta finestra, a l'apartat *Code Assistance* cal afegir totes les rutes dels directoris dels fitxers d'inclusions i indicar-li quines paraules ha d'interpretar com a macros. Això serveix per a a facilitar l'escriptura del codi font a l'editor, no es obligatori però si que facilita més la feina durant el desenvolupament. Aquestes opcions s'han d'afegir tant per a C com per a C++ tal i com es mostra a les imatges A.15 i A.16.

L'última tasca que queda per fer en aquesta finestra és la d'incloure les extensions dels fitxer de codi font CUDA C/C++ a l'IDE. Per fer-ho, cal anar al darrer apartat, *Other* i modificar els camps *C File Extensions* i *C/C++ Header File Extensions* per tal que quedin com mostra la figura A.17.

En aquest moment ja estan configurades les opcions globals, però resta configurar el projecte. A Netbeans, les aplicacions desenvolupades en CUDA C/C++ són creades com si fossin un projecte per a crear una aplicació C/C++ escollint aquesta opció a la finestra de nou projecte tal i com mostra la imatge A.18.

Quan es clica a *Next*, apareixen les opcions bàsiques del nou projecte, com el nom i el directori on es guardarà. Cal tenir en compte que s'ha de desactivar la casella *Create Main File* perquè això crea un fitxer d'extensió *.cpp* que no s'usarà, i a més a més, cal seleccionar la col·leció d'eines *NVCC* creada anteriorment tal i com mostra la figura A.19.

Una vegada creat el projecte cal configurar les seves propietats, per fer-ho cal escollir l'opció *Properties* del menú contextual que apareix al clicar amb el botó dret del ratolí sobre el projecte, això farà que aparegui la finestra de propietats. Ara cal modificar els camps *Include Directories* i *Additional Options* dels apartats *C Compiler* i *C++ Compiler* afegint el directori dels fitxers d'inclusions del SDK (*../../NVIDIA_GPU_Computing_SDK/C/common/inc*) al primer, i les opcions `-DUNIX --compiler-options -fno-strict-aliasing` al segon. També cal comprovar que el camp *Tool* tingui el valor `nvcc`. La imatge A.20 mostra com queda la llista d'opcions en el cas del compilador de C.

També cal modificar la configuració de l'enllaçador, que ha de quedar com mostra la imatge, modificant els camps *Additional Library Directories* i *Libraries*.

Ara ja està configurat el projecte correctament pel seu desenvolupament dins Netbeans. Encara que cal tenir en compte que es poden configurar els arguments d'execució del programa al camp *Arguments* de l'apartat *Run* dintre les opcions del projecte. I per compilar i executar, només cal pitjar F11 i F6, o fer-ho a través del menú contextual del projecte amb les opcions *Build* i *Run* com mostra la figura A.22.

En el cas de la biblioteca libvpx, el programa principal és un projecte C/C++ al qual s'enllaça la biblioteca CUDA desenvolupada que no és més que un projecte configurat com un projecte CUDA C/C++ amb la diferència que en el moment d'escollir el tipus del nou projecte s'ha escollit l'opció *C/C++ Static Libray*. Només cal afegir la ruta a l'arxiu *.a* amb la biblioteca estàtica creada al camp *Libraries* de l'enllaçador del projecte *simple_decoder_lite*, encara que aquesta aplicació també fa ús de les biblioteques *Posix Threads* i *Mathematics*, que són biblioteques estàndards disponibles per defecte a Netbeans.

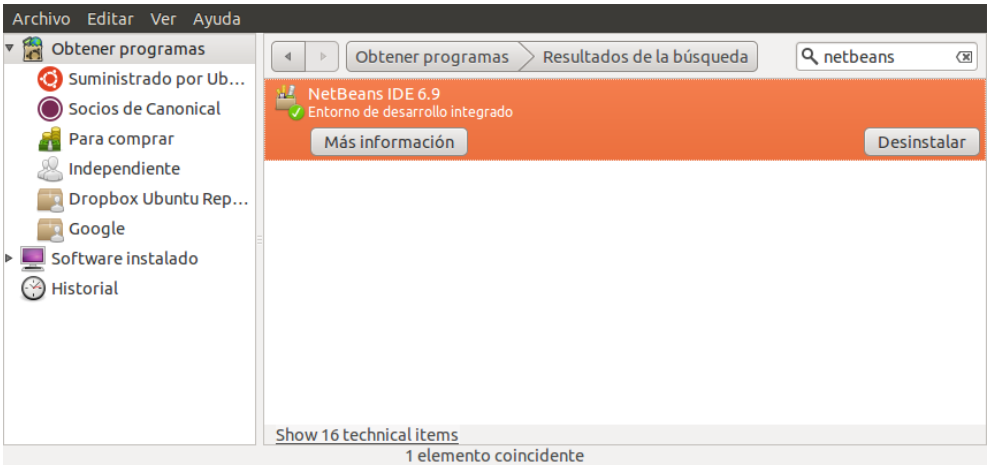

Figura A.12: Netbeans al Centre de Software d'Ubuntu

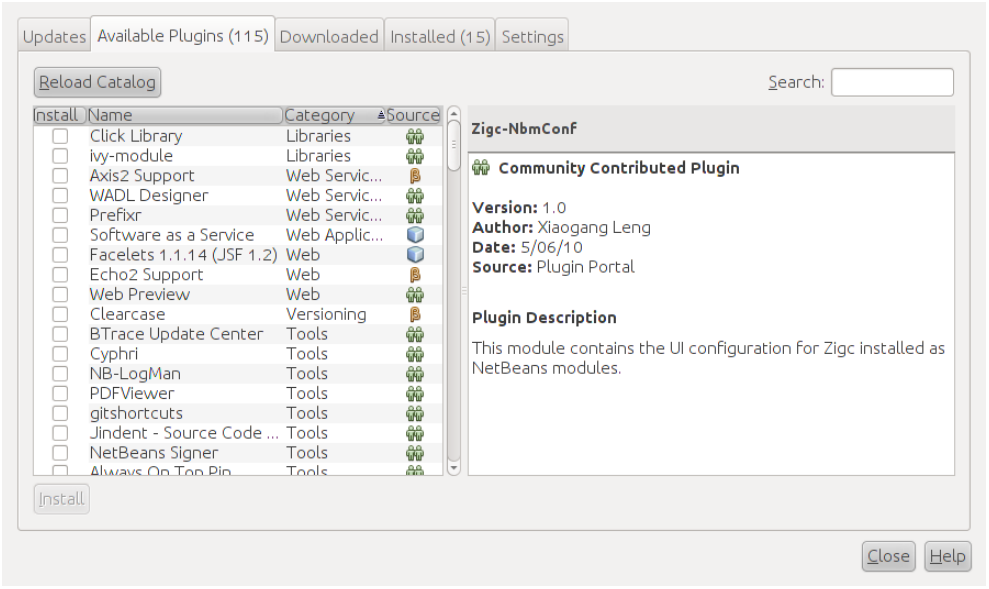

Figura A.13: Gestor de *plugins* de Netbeans 6.9

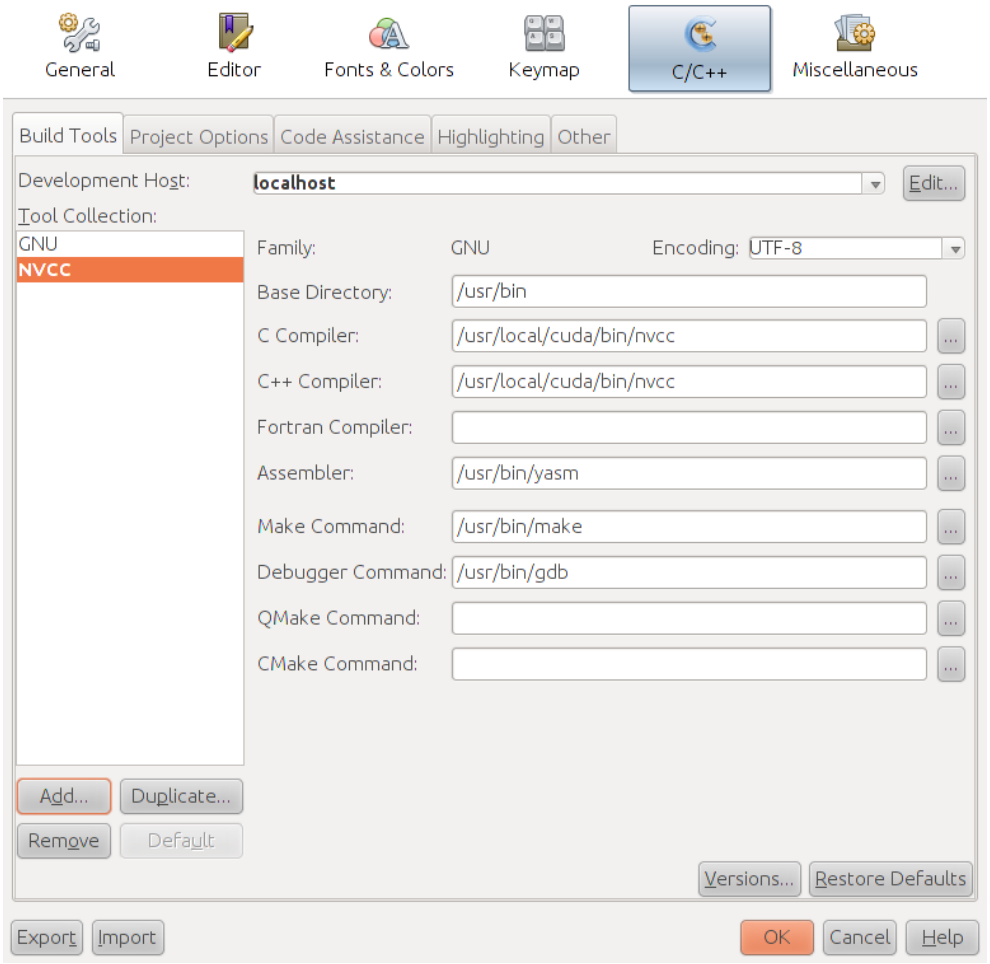

Figura A.14: Configuració de la col·lecció d'eines per a CUDA

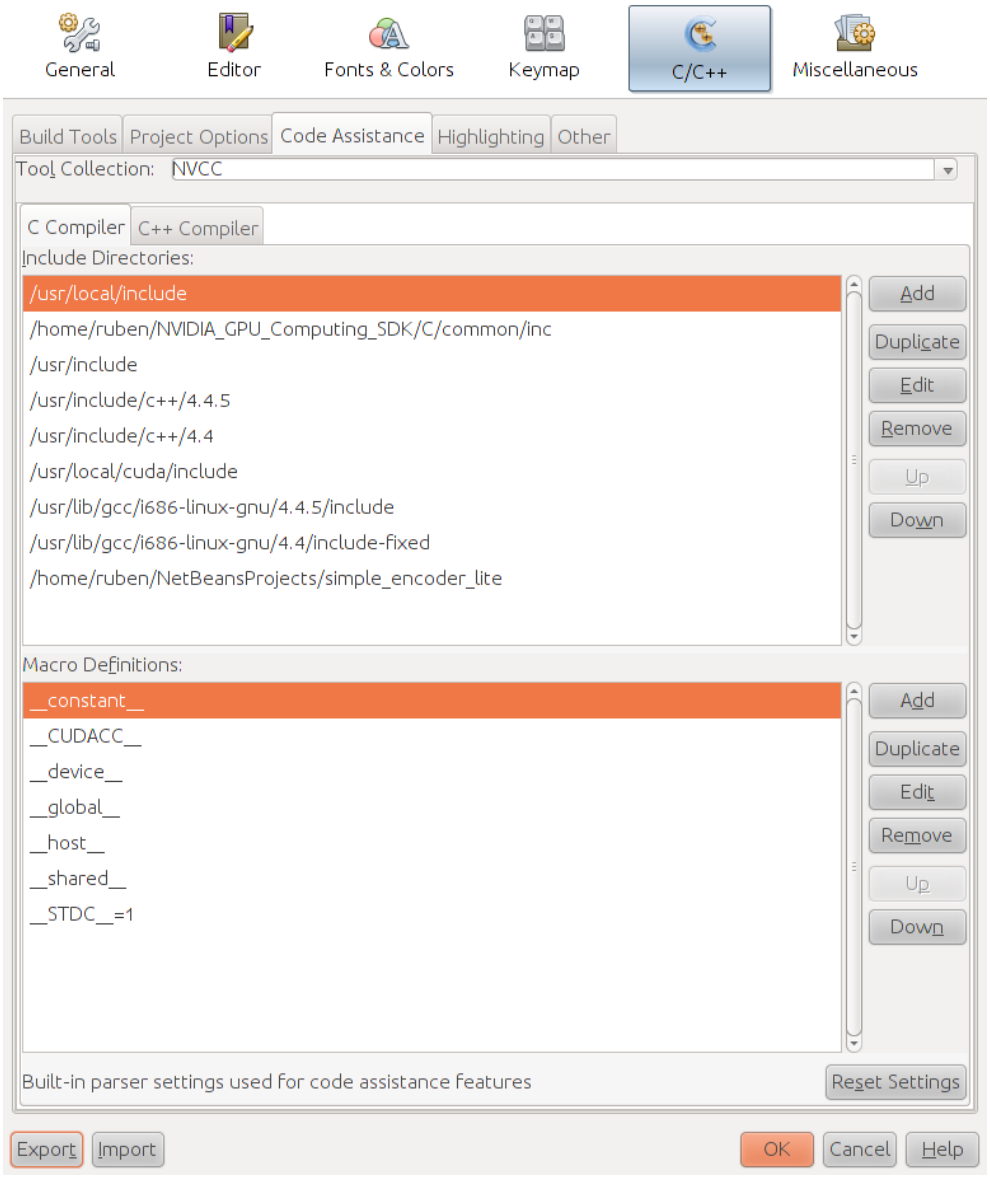

Figura A.15: Configuració de l'assistent de codi per a C

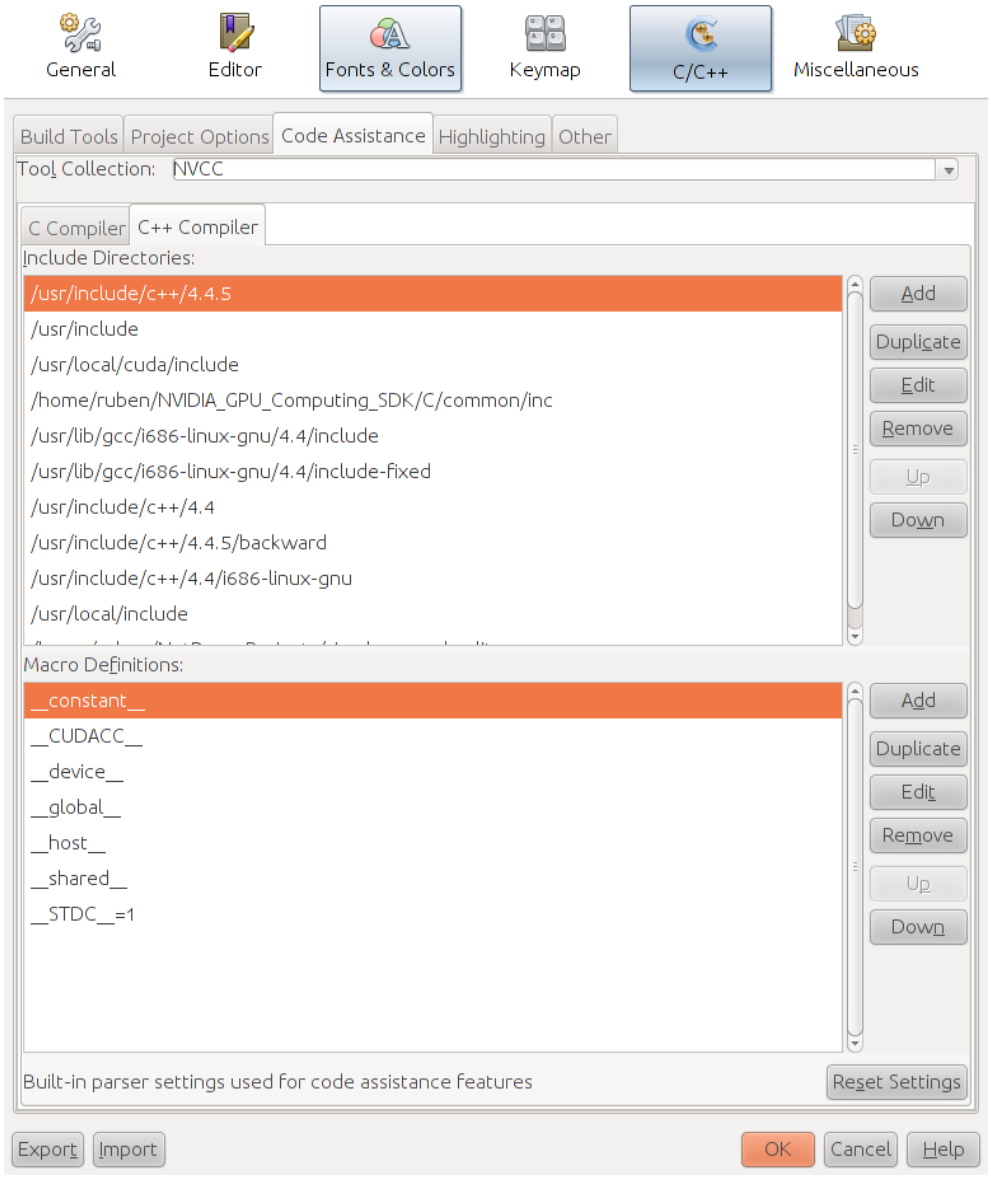

Figura A.16: Configuració de l'assistent de codi per a C++

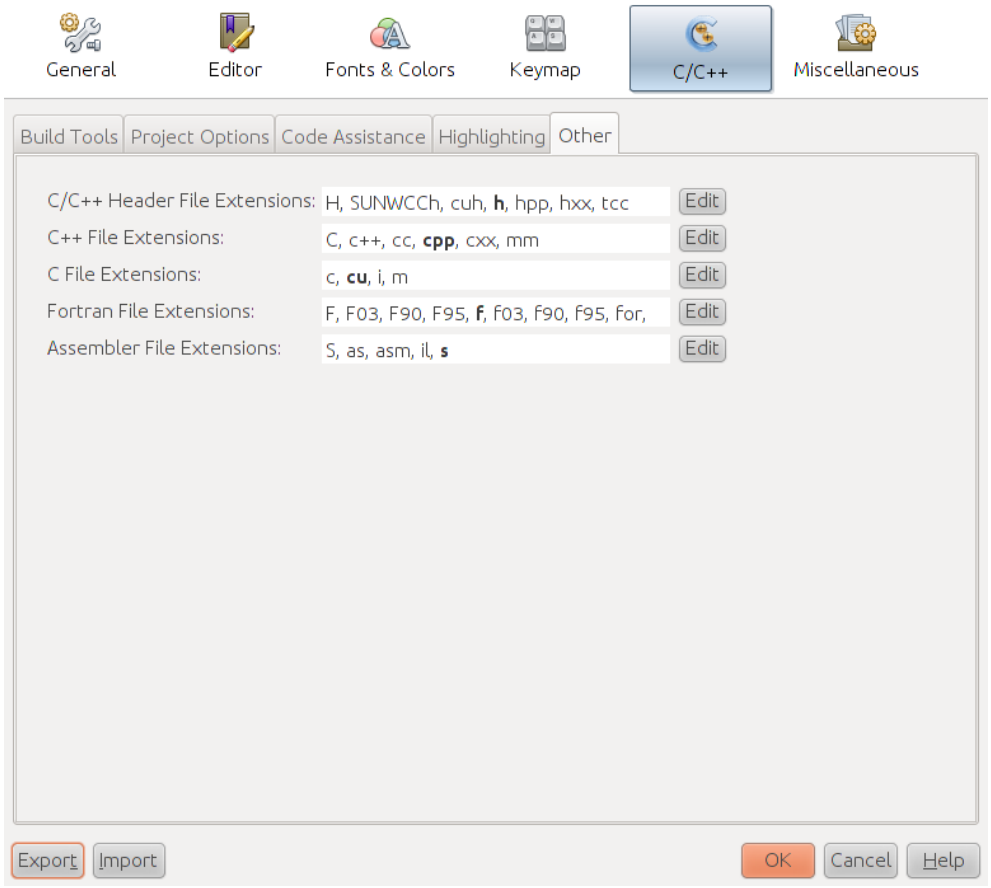

Figura A.17: Opcions de les extensions de fitxer suportades

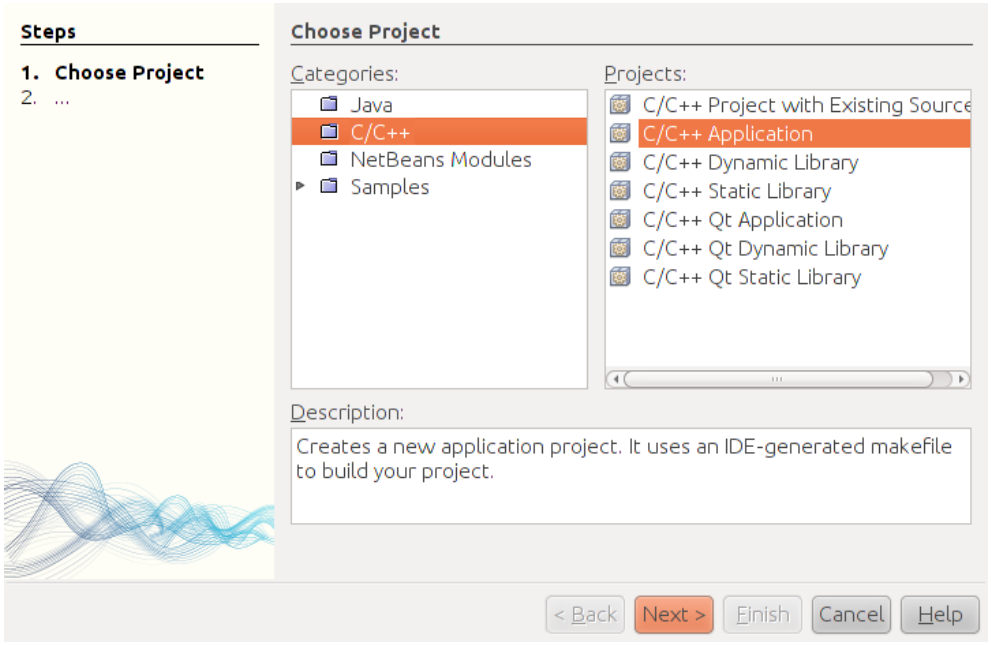

Figura A.18: Creació d'un projecte per a una aplicació C/C++ a Netbeans

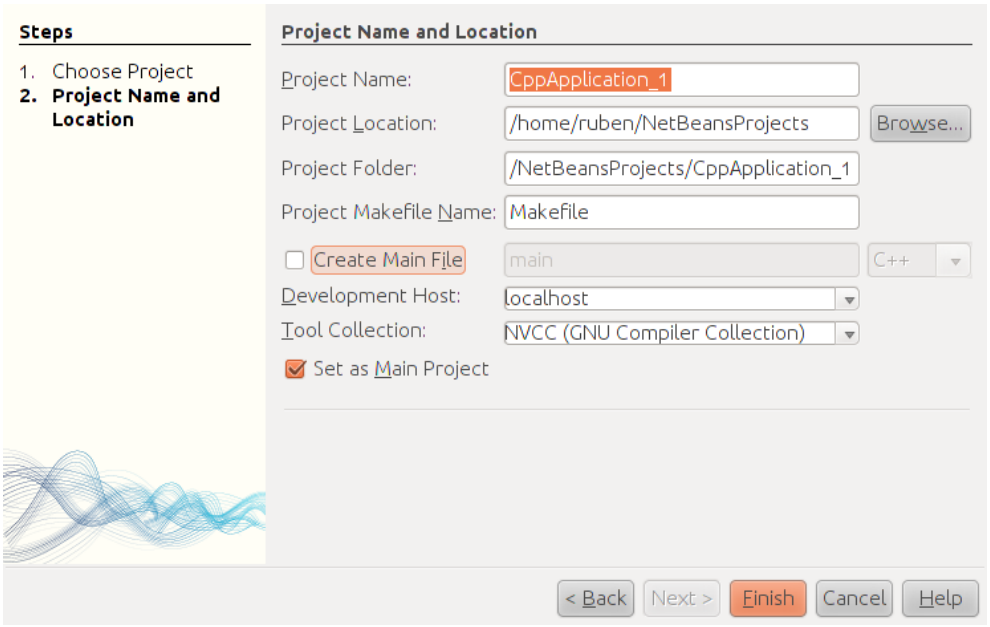

Figura A.19: Opcions bàsiques del nou projecte CUDA C/C++

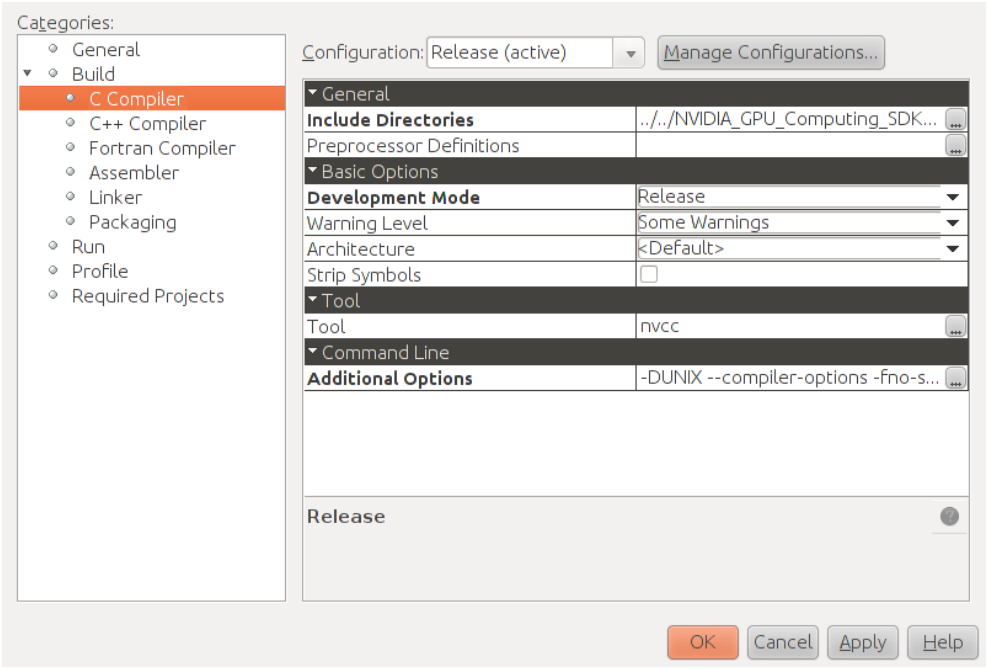

Figura A.20: Llista d'opcions per el compilador C del projecte CUDA

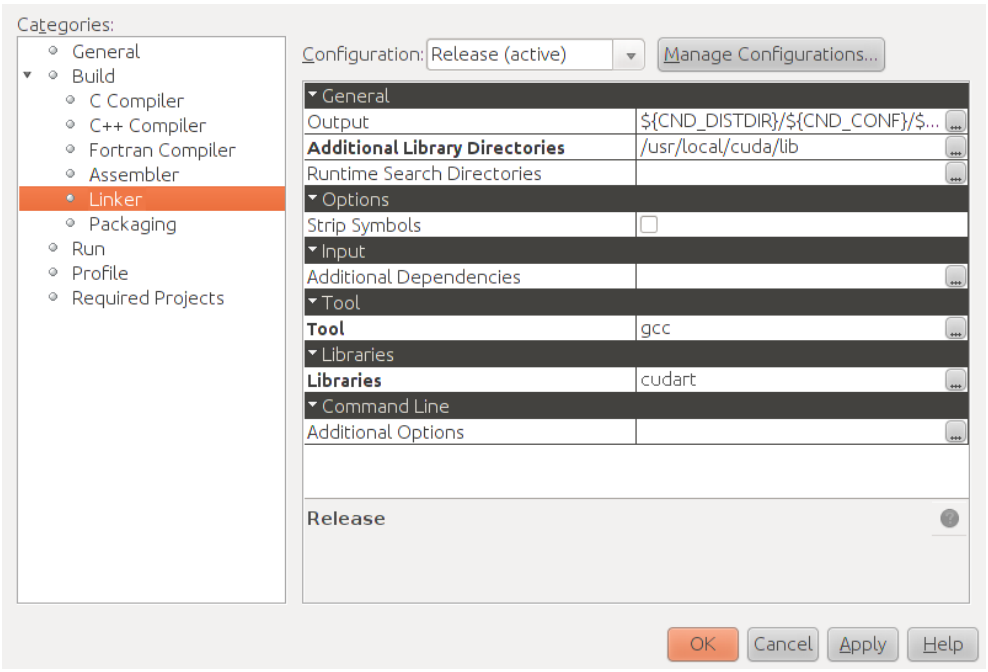

Figura A.21: Opcions de l'enllaçador per a un projecte CUDA C/C++

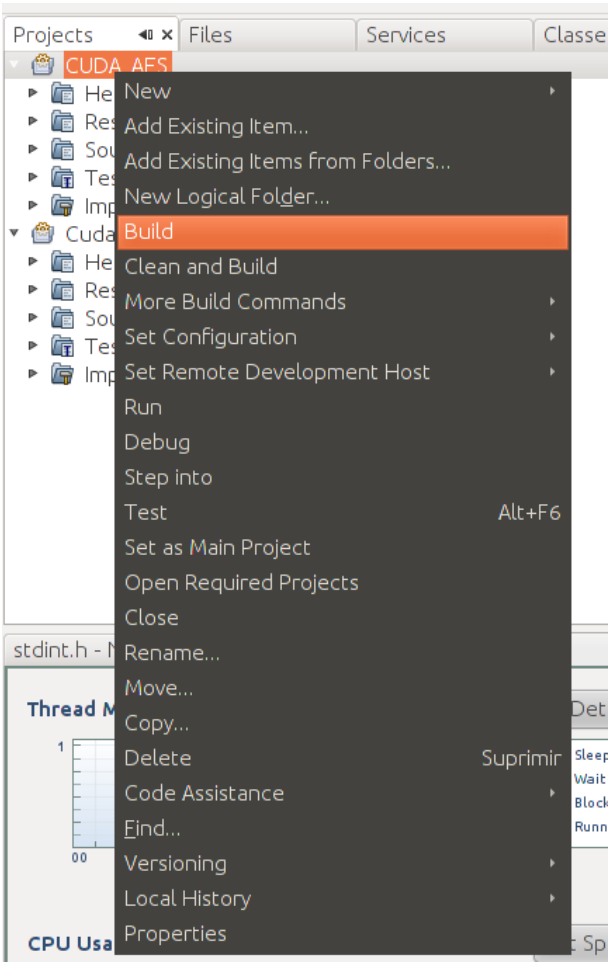

Figura A.22: Menú contextual del projecte

Apèndix B

Manuals de l'usuari

B.1 Xifrador AES

B.1.1 Paràmetres de l'execució

Els paràmetres de l'execució del xifrador AES són els següents:

1. Mode: Aquest pot ser `e` si es vol xifrar l'arxiu d'entrada, o `d` si es vol desxifrar.
2. Longitud de la clau: Aquest paràmetre representa la longitud de la clau usada al algorisme AES. Pot tenir qualsevol d'aquests tres valors:
   - `1`: Indica una clau de 128 bits.
   - `2`: Indica una clau de 192 bits.
   - `3`: Indica una clau de 256 bits.
3. Arxiu d'entrada
4. Arxiu de sortida
5. Arxiu que conté la clau del xifrat.

Un exemple d'execució del xifrador és:

```
$>./AES e 3 in out key
```

B.1.2 Sortida per pantalla

Durant l'execució del xifrador, la següent informació s'imprimeix per la consola de comandes:

1. Si s'ha pogut obrir els tres fitxers correctament.
2. Si s'ha pogut llegir la clau del fitxer i quin és el seu contingut.
3. Si s'ha expandit la clau i inicialitzat AES correctament.
4. Si és la versió GPU, si s'ha copiat la clau expandida a la GPU correctament.
5. Per cada vegada que s'omple el buffer:
   (a) Si s'ha carregat correctament
   (b) Si les dades s'han tractat correctament

(c) Si les dades tractades s'han escrit correctament

6. Al finalitzar el tractament del fitxer mostra:

   (a) Un avís de finalització

   (b) El nombre de bytes tractats

   (c) El nombre d'accessos a disc

Un exemple de com es mostra aquesta informació és l'execució que es mostra a la figura B.1.

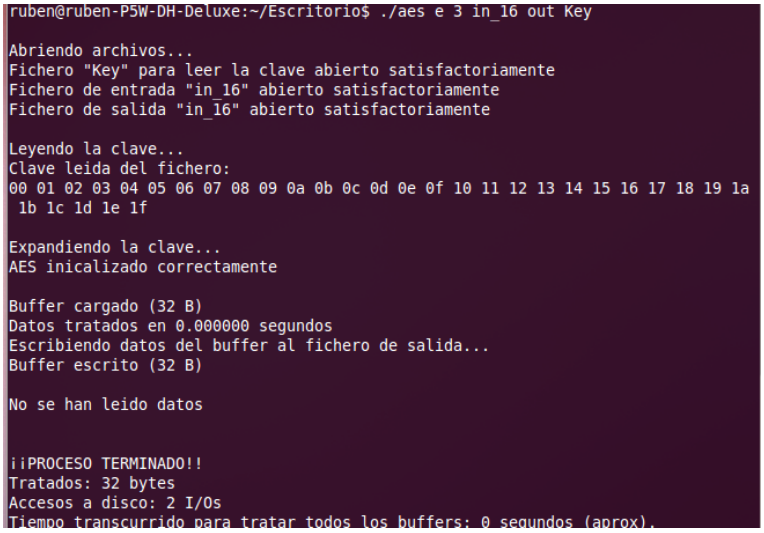

Figura B.1: Exemple de la sortida per pantalla del xifrador AES

B.1.3 Missatges d'error

El xifrador AES pot retornar diferents missatges d'error, són els següents:

- Si el mode és diferent de `d` i de `e`, mostra un missatge d'error i finalitza l'execució del programa.
- Si la mida de la clau no és `1`, `2`, o `3`, mostra un missatge d'error i finalitza l'execució del programa.
- Si no es pot obrir qualsevol dels tres arxius, mostra un missatge d'error i finalitza l'execució del programa.
- Si la longitud de la clau llegida és diferent a la longitud de la clau indicada al segon paràmetre, mostra un missatge d'error i finalitza l'execució del programa.
- Si el fitxer d'entrada i el de sortida és el mateix, mostra un avís per pantalla i canvia el nom del fitxer de sortida afegint l'extensió *.rsc* a l'arxiu de sortida.
- Si mentre llegeix o escriu al disc dur es produeix un error, mostra un missatge d'error i finalitza l'execució del programa.
- Si en la versió GPU qualsevol de les tasques de gestió de memòria retorna error, mostra un missatge d'error i finalitza l'execució del programa.

B.2 FileGenerator

Aquest programa és el que s'ha fet servir per a generar els arxius que s'han utilitzat per a fer els anàlisis. Els paràmetres que espera són:

1. Fitxer origen dels 16 bytes usats com a mostra.
2. Carpeta on es s'emmagatzemen els arxius generats.

Aquest programa és molt simple, llegeix els 16 bytes de l'arxiu de mostra que s'interpreten com a una matriu *State* del xifrador AES i genera arxius de 128, 256, 384, i 512MB. Aquests arxius contenent repeticions dels 16 bytes del fitxer origen. Cal tenir en compte que aquest programa no mostra cap tipus d'informació per pantalla ni fa la gestió de cap tipus d'error.

B.3 Scripts de test

Els dos scripts que s'han usat per a fer els tests de rendiment no accepten cap paràmetre i l'única informació que mostren per pantalla és la ruta de l'últim arxiu que s'ha començat a processar.

Glossari

API	Application Programming Interface o Interficie de Programació d'Aplicacions. En el desenvolupament del software s'usen aquestes interficies per fer que una aplicació interaccioni i faci ús de les funcionalitats d'un sistema aliè a l'aplicació. L'accés a aquestes funcionalitats es fan mitjançant la crida a les funcions definides a la API.
APP	Parallel Processing o Processament en Paral·lel Accelerat.
C99	És un dialecte modern de C. Basat en C90, la versió anterior, implementa noves característiques al llenguatge i millora l'us que fa el llenguatge de les noves tecnologies hardware.
Consola	És l'interpret de comandes del sistema. L'usuari pot comunicar ordres directes mitjançant una interfície de text pla i aquest li mostra els resultats corresponents per pantalla.
CUDA	Compute Unified Device Architecture o Arquitectura de Dispositiu de Computació Unificada.
Còdec de video	És una biblioteca de software que permet interpretar un format de vídeo comprimit per a que un reproductor de video el pugui mostrar per pantalla.
DMA	És l'acrònim d'Acès Directe a Memòria. Aquesta tecnologia permet l'intercanvi de les dades directament entre la memòria principal (RAM) i el disc dur sense haver de passar per la CPU. Això fa que disminueixi el temps de la transferencia de les dades entre RAM i disc dur.
Framework	És conjunt d'eines per a desenvolupar un software d'una manera més fàcil i amb més comoditat.
GF()	S'interpreta com a camp finit o camp de Galois (Galois Field). És un cos que conté un nombre finit d'objectes. S'utilitzen en criptografia per resoldre el problema del sobreeiximent dels tipus de dades, ja que a criptografia s'usen nombres molt grans.
GPGPU	General Purpouse Graphic Processing Unit o Unitat de Processament de Gràfics de Proposit Genèric. S'anomena així a aquelles GPU que poden fer tasques més genériques a més a més de les específiques d'una GPU, encara que no arriben a ser unes tasques tan genériques com les d'una CPU.
GPU	Graphic Processing Unit o Unitat de Processament de Gràfics. És un element hardware especialitzat en transformar tota la informació que li arriba en una interfície gráfica que es presenta a l'usuari. En termes més simples, aquest element dibuixa per pantalla la informació que li lliura el sistema.

GPU — Graphic Processing Unit o Unitat de Processament de Gràfics. És un element hardware especialitzat en transformar tota la informació que li arriba en una interfície gráfica que es presenta a l'usuari. En termes més simples, aquest element dibuixa per pantalla la informació que li lliura el sistema.

GUB — És l'acronim de GNU GRand Unified Bootloader. En cas de tenir-lo instal·lat, l'ordinador no inicia cap sistema operatiu, sino que s'inicïa GUB. Aquest software permet escollir quin sistema operatiu iniciar en cas d'existir més d'un instal·lat a l'ordinador. En el cas d'Ubuntu, permet escollir amb quina versió del kernel dels que hi ha instal·lats arrencar la distribució.

GUI — Graphic User Interface o Interfície Gràfica d'Usuari. És un conjunt d'imatges que representen l'informació i les accions disponibles per a interactuar amb ella.

IDE — Integrated Development Environment o Entorn de Desenvolupament Integrat. És un software que, com el seu nom ja indica, integra diverses eines per al desenvolupament d'aplicacions per un o més llenguatges de programació. Les eines més comunes que podem trobar a un IDE són:

- Editor de codi
- Compilador
- Eines de depuració de codi
- Framework

IVF — És el format Indeo Video Format, més conegut com a Intel Indeo (Intel va desenvolupar aquest codec de video). Aquest còdec va ser bastant popular durant els any 90 del segle XX però va ser superat pels estàndars MPEG i avui en dia s'utilitza molt poc i no tots els reproductors de video els suporten.

JPEG — Joint Photographic Experts Group és el nom de l'entitat que va crear l'estàndar de compressió d'imatges estàtiques que rep el mateix nom. Aquest format, és un dels més utilitzats en la digitalització d'imatges.

KeyFrame — Fotograma clau. També rep el nom d'Intraframe.

Macro — Una macro es un tros de codi al cual se li dona un nom, i n'hi ha de dos tipus: macros semblants a objectes i macros semblants a funcions. Allà on es nombrada una macro, el compilador insereix el codi definit amb el nom de la macro usat.
L'ús de macros és semblant a l'ús de funcions, però les macros s'executen més ràpidament encara que al substituir el codi allà on es nombren provoca que l'executable resultant ocupi més espai. El benefici d'usar macros apareix quan aquesta conté poc codi i s'usa poques vegades. Si cal usar una macro moltes vegades, es recomana usar una funció en lloc seu, ja que l'executable final ocuparà menys espai.

MPEG — Són una serie de estàndars de video definits pel Moving Picture Experts Group (MPEG). Avui en dia aquests estàndars son molt usats per la captura de vídeo.

OpenCL	Open Computing Language o Llenguatge de Computació Obert.
OpenCL	Open Computing Language o Llenguatge de Computació Obert.
Renderitzar	És el procés de representar en un dispositiu de sortida d'un ordinador (una imatge o una escena), generalment en tres dimensions, simulant-ne els efectes òptics de llum, ombra, color, textura o moviment a partir de les dades d'un model computacional.
Script	És un programa per a la consola de comandes. S'usa per a automatitzar una serie de comandes molt llargua i repetitiva i així alleugerir la seva execució per part de l'usuari. A Ubuntu s'usa el terminal Bash, que té el seu propi llenguatge per els scripts. Tot el que es pot fer en una consola de comandes es pot automatitzar amb script.
TUR	Acrònim de Tarifa d'Últim Recurs. És la tarifa que imposa l'estat per al cost de l'energia a totes les comercialitzadores del país.
Wrapper	Es pot traduir com a "embolcall". En el desenvolupament del software, per fer que un element software tingui més funcionalitats sense alterar-lo, es crea un software que interactúa amb el primer i que, basant-se en les funcionalitats d'aquest en crea de noves.
XOR	Operació binaria, OR exclusiva. És a dir, si els bits comparats són iguals, el resultat és 0, altrament el resultat és 1.
YUV	És un format de video que conté aquest sense cap mena de compressió ni cap tipus de dada sobre el medi que conté.

Bibliografia

[1] Oracle Corporation and/or its affiliates. Netbeans ide. Consultat: 16/10/2011.

[2] BSC. El bsc, nombrado cuda center of excellence. Consultat: 17/11/2011.

[3] NVIDIA corp. Cuda toolkit 4.0 (may 2011). Consultat: 16/10/2011.

[4] Col·laboradors de la Viquipèdia. Llei de moore. Consultat: 30/09/2011.

[5] Col·laboradors de la Viquipèdia. Tianhe-i. Consultat: 25/09/2011.

[6] Col·laboradors del projecte Viquipedia. Advanced encryption standard. Consultat: 12/09/2011.

[7] Col·laboradors del projecte Viquipedia. Modes d'operació dels sistemes de xifratge per blocs. Consultat: 12/09/2011.

[8] Inc. Elemental Technologies. Badaboom. Consultat: 10/10/2011.

[9] isnull. Tutorial cuda 3.2 and visual studio 2008: The first cuda program! Consultat: 16/10/2011.

[10] Edward Kandrot Jason Sanders. *CUDA by example: An introduction to General-Purpose GPU programming*. Addison-Wesley. Consultat: 15/09/2011.

[11] DI Management Services Pty Limited. Using padding in encryption. Consultat: 20/09/2011.

[12] Victor Lo. A begginers guide for mpeg-2 standard. Consultat: 13/09/2011.

[13] Canonical Ltd. Ubuntu 10.10 (maverick meerkat). Consultat: 15/10/2011.

[14] Microsoft. Visual studio 2008 express developer center -> downloads. Consultat: 16/10/2011.

[15] Microsoft. Windows 7. Consultat: 16/10/2011.

[16] NVIDIA. *NVIDIA CUDA C Programming Guide*. NVIDIA. Consultat: 1/10/2011.

[17] University of California. Run seti@home on your nvidia gpu. Consultat: 1/10/2011.

[18] National Institute of Standards and Technology (NIST). *Specification for the ADVANCED ENCRYPTION STANDARD (AES)*. National Institute of Standards and Technology (NIST). Consultat: 5/10/2011.

[19] VideoLAN Organization. x264 source code. Consultat: 12/09/2011.

[20] Vijay Pande and Stanford University. Folding@home: distributed computing. Consultat: 25/09/2011.

[21] The WebM Project. Documentation. Consultat: 20/09/2011.

[22] Processor review. Intel core i7 920 review benchmark and overclocking. Consultat: 15/12/2011.

[23] rxwen. View raw yuv file with mplayer. Consultat: 25/10/2011.

[24] ENDESA S.A. ¿Qué es la TUR? Consultat: 20/12/2011.

[25] William Stallings. *Cryptography and Network Security Principles and Practices*. Prentice Hall. Consultat: 20/09/2011.

[26] Tommy. fopen() file size limitation. Consultat: 20/12/2011.

[27] Damien Triolet. Nvidia cuda : practical uses. Consultat: 20/12/2011.

[28] Parth Trivedi. How to build or open a cuda project in the graphical development environment netbeans. Consultat: 16/10/2011.

[29] Xiph.org. Xiph.org test media. Consultat: 16/10/2011.

[30] Xvid. Xvid. Consultat: 13/09/2011.